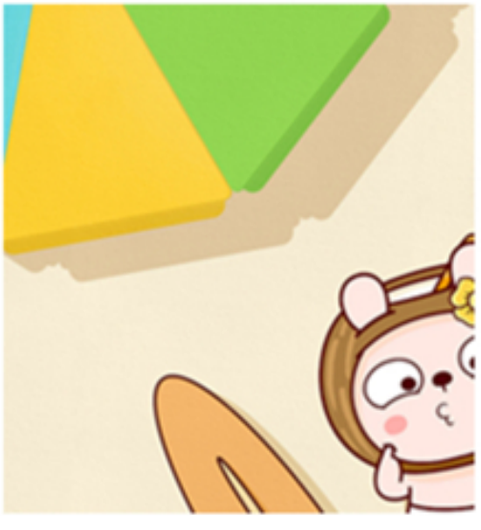
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

Docker容器技术与应用

版权所有，严禁传播，违者自负法律责任！

主　编　朱晓彦　聂　哲

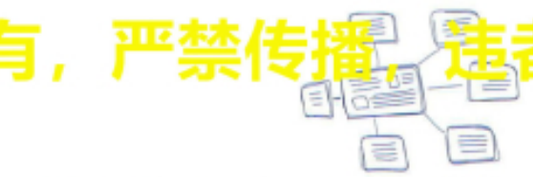
副主编　刘学普　郑美容　池瑞楠

主　审　王　海　张　昕

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！



非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

高等教育出版社

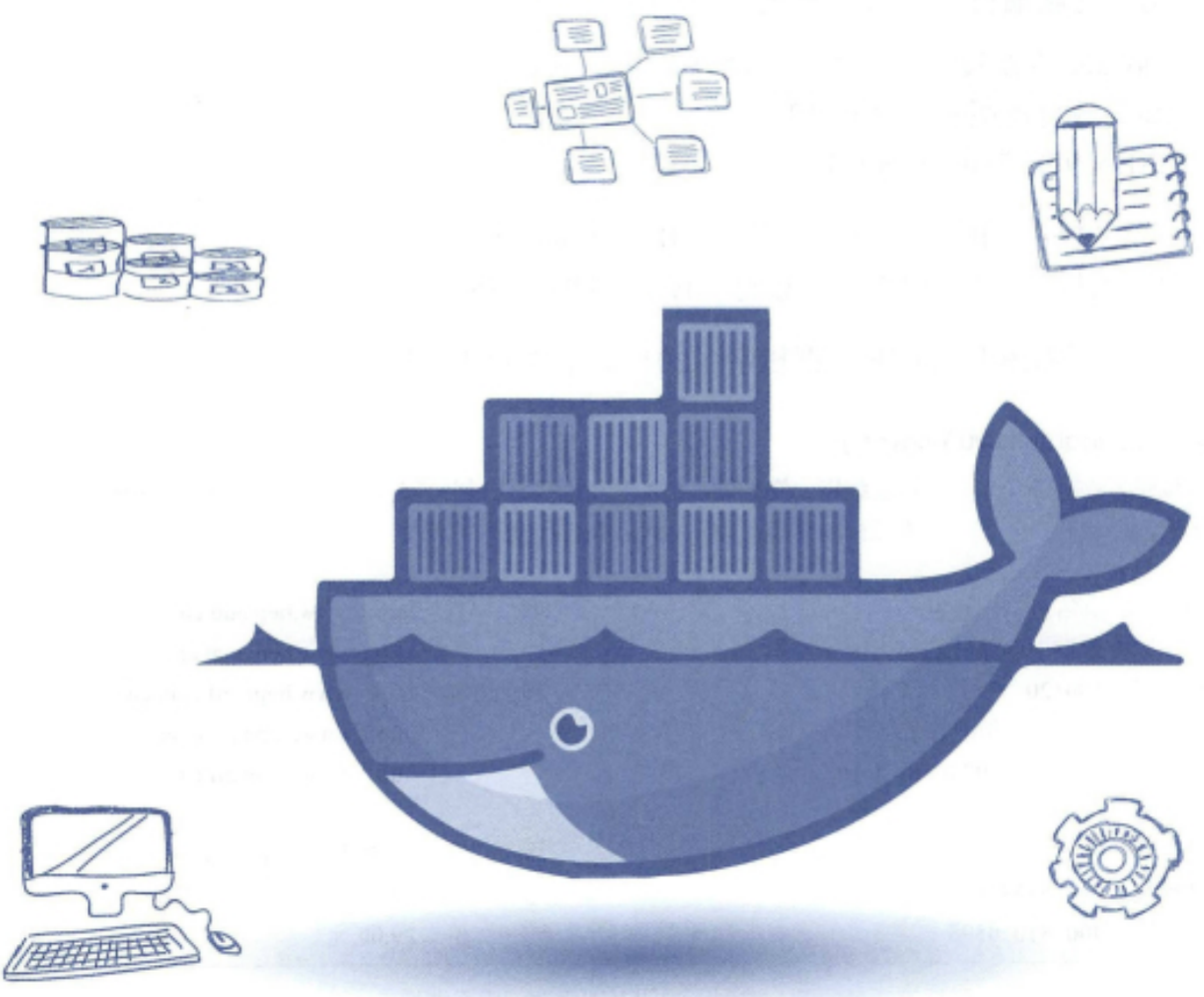
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

CETC 云计算技术与应用专业校企合作系列教材

Docker
容器技术与应用

主　编　朱晓彦　聂　哲
副主编　刘学普　郑美容　池瑞楠
主　审　王　海　张　昕



高等教育出版社·北京

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

内容摘要

本书是云计算技术与应用专业校企合作系列教材。

在云计算时代，企业通过 OpenStack 私有云、公有云等平台将应用从硬件迁移到云上，解决了硬件管理和运维的问题，然而软件配置、管理和运维的问题依然存在，Docker 的出现正好能帮助软件相关从业者开阔思路，尝试新的软件管理方法来解决这个问题。通过掌握 Docker 技术，开发人员可以实现持续集成 CD，运维人员可以实现持续交付 CI，享受先进的自动化运维理念和工具，无须运维人员介入即可顺利运行于各种运行环境。

本书从 Docker 的基本使用入手，深入浅出地讲解了 Docker 的构建、操作、技术原理和实际使用过程中的典型项目和案例，内容较全面，分为 7 个单元，前两个单元为 Docker 容器的安装、Docker 容器的使用和管理，介绍了容器构建和使用基础；第 3、4、5 单元为 Docker 仓库 Registry，Docker 网络、存储和接口，Docker 容器编排与集群，介绍了容器的高级功能；第 6、7 单元为容器服务管理平台 Rancher 和使用 Docker 构建持续集成，以案例和项目的形式，展现给学习者真实的应用场景。

本书适用于本科和高职的计算机网络技术、网络工程、云计算技术与应用等相关方向或专业的学生学习 Docker 技术，同时也适用于企业开发和运维人员快速掌握 Docker 的基本技术及其应用。

图书在版编目（CIP）数据

Docker 容器技术与应用 / 朱晓彦，聂哲主编. --北京：高等教育出版社，2017.12

ISBN 978-7-04-048943-9

Ⅰ. ①D… Ⅱ. ①朱… ②聂… Ⅲ. ①Linux 操作系统－程序设计－高等职业教育－教材 Ⅳ. ①TP316.85

中国版本图书馆 CIP 数据核字（2017）第 280641 号

Docker Rongqi Jishu yu Yingyong

策划编辑	许兴瑜	责任编辑	许兴瑜	封面设计	赵 阳	版式设计	徐艳妮
插图绘制	杜晓丹	责任校对	吕红颖	责任印制	赵义民		

出版发行	高等教育出版社	网 址	http://www.hep.edu.cn
社 址	北京市西城区德外大街 4 号		http://www.hep.com.cn
邮政编码	100120	网上订购	http://www.hepmall.com.cn
印 刷	大厂益利印刷有限公司		http://www.hepmall.com
开 本	787mm × 1092mm 1/16		http://www.hepmall.cn
印 张	11.75		
字 数	320 千字	版 次	2017 年12月第 1 版
购书热线	010-58581118	印 次	2017 年12月第 1 次印刷
咨询电话	400-810-0598	定 价	29.00 元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换

版权所有 侵权必究

物 料 号 48943-00

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

前　言

一、缘起

Docker 最初是 DotCloud 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目。它是基于 DotCloud 公司多年的云服务技术的一次革新，并于2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在 GitHub上进行维护。Docker 项目后来还加入了 Linux 基金会，并成立推动开放容器联盟。

Docker 使用 Google 公司推出的Go 语言进行开发实现，基于 Linux 内核的Cgroups、NameSpace 及AUFS 类的Union FS等技术对进程进行封装隔离，属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其他隔离的进程，因此也称其为容器。最初实现是基于 LXC 的，从 Docker 0.7 以后开始去除 LXC，转而使用自行开发的 ibcontainer，从 Docker 1.11 开始，则进一步演进为使用RunC和Containerd。

学习 Docker 的源码并不是一个枯燥的过程，读者可以从中理解 Docker 架构的设计原理。Docker 对使用者来讲是一个 C/S 模式的架构，Docker 的后端是一个非常松耦合的架构，模块各司其职，并有机组合，支撑 Docker 的运行。Docker 的框架包括 Docker 后台进程 Doemon、存储数据卷、网络、镜像仓库、镜像、容器实例和控制台等。

二、结构

本书教学内容采用模块化的编写思路，将 Docker 容器、容器服务平台 Rancher、持续集成 DevOps 三方面的学习内容分为了 Docker 容器的安装，Docker 的使用和管理，Docker 仓库 Registry，Docker 网络、存储和接口，Docker 容器编排与集群、容器服务管理平台 Rancher，使用 Docker 构建持续集成 7 个教学单元和 20 个教学任务。

每个单元通过学习目标引出单元的教学核心内容，明确教学任务。每个任务的编写分为任务描述、知识学习、任务实施、项目实训 4 个环节。

- 任务描述：简述任务目标，展示任务实施效果，提高学生学习兴趣。
- 知识学习：详细讲解知识点，通过系列实例实践，边学边做。
- 任务实施：通过任务综合应用所学知识，提高学生系统地运用知识的能力。
- 项目实训：在任务实施的基础上通过“学、仿、做”达到理论与实践的统一、知识内化的教学目的。

最后进行单元小结，总结本单元的教学重点、难点。

三、特点

1. 针对性强，内容选取以实用为主

本书以云计算技术专业学生的就业岗位群为导向，整个课程分为两大部分：知识学习和技术应用。知识学习以 Docker 概述、容器技术的基本概念、容器管理平台 Rancher 的简介等基本知识为主，培养学生具有较为系统的 Docker 容器技术基本技能；技术应用以容器的基本使用、管理平台的运维、持续集成的构建为主，内容

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

设计比较丰富，便于学生理解和掌握。

2. 精心设计，教学内容与数字化资源有机结合

本书以教学内容为主线将各项数字化资源有机结合在一起，形成完整的数字课程。

数字化资源包括 3 个方面的内容：第一、课程本身的基本信息，包括课程简介、学习指南、课程标准、整体设计、单元设计等；第二、教学内容中重难点的微课视频教学资源，既方便课内教学，又方便学生课外预习和复习；第三、课程项目实训，这包括课程的重难点剖析，循序渐进的综合项目开发案例、实训任务单等。

本书内容满足课堂教学的需要，而数字化资源为学生课外自主探究学习提供了一个良好的平台，课堂教学与智慧职教平台结合，提高了教学效果。

四、使用

1. 教学内容课时安排

本书建议授课 58 学时，教学单元与课时安排见表 1。

表 1　教学单元与课时安排

序号	单元名称	学时安排
1	Docker 容器的安装	4
2	Docker 的使用与管理	8
3	Docker 仓库 Registry	10
4	Docker 网络、存储和接口	10
5	Docker 容器编排与集群	10
6	容器服务管理平台 Rancher	10
7	使用 Docker 构建持续集成	12
课时总计		64

2. 课程资源一览表

本书是云计算技术与应用专业校企合作系列教材，开发了丰富的数字化教学资源，可使用的教学资源见表 2。

表 2　课程教学资源一览表

序号	资源名称	表现形式与内涵
1	课程简介	Word 文档，包括对课程内容的简单介绍和对课时、适用对象等项目的介绍，让学习者对 Docker 有个简单的认识
2	学习指南	Word 文档，包括学前要求、学习目标要求及学习路径和考核标准要求，让学习者知道如何使用资源完成学习
3	课程标准	Word 文档，包括课程定位、课程目标要求、课程内容与要求，可供教师备课时使用
4	整体设计	Word 文档，包括课程设计思路、课程具体的目标要求、课程内容设计和能力训练设计，同时给出考核方案设计，让教师理解课程的设计理念，有助于教学实施
5	授课 PPT 和视频	PPT 文件和 AVI 视频文件，可帮助教师理解如何教好 Docker 这门课
6	教学单元设计	Word 文档，分任务给出课程教案，帮助教师完成一堂课的教学细节分析
7	微课	AVI 视频文件，有助于学习者更加直观的学习

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

续表

序号	资源名称	表现形式与内涵
8	电子课件	PPT 文件，使用 PowerPoint 2003 版，也可根据教师的实际需要进行修改后使用
9	实训任务单	Word 文档，加深对任务设计实训的学习，并给出实训的详细实训步骤
10	案例	Tar 包，包括单元项目案例和综合案例，综合运用所学的知识
11	习题库、试卷库	Word 文档，习题包括理论习题和操作习题，试卷包括单元测试和课程测试。通过习题和测试，学习者可加深对知识的掌握程度
12	附书源码	Tar 包，包括本书中的所有例题和任务源代码

教师可发邮件至邮箱 douxue@1daoyun.com 索取教学基本资源。

五、致谢

本书由安徽工业经济职业技术学院的朱晓彦，以及南京第五十五所技术开发有限公司、江苏省一道云科技发展有限公司的工程师共同编写，在此对各位参编老师、工程师致以衷心的感谢。

由于作者水平有限，错误和不足之处在所难免，恳请各位读者给予批评、指正，编者将不胜感激。

编者

2017 年 11 月

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：

1、书籍版权归著者和出版社所有

2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用

4、本PDF获得者不得在互联网上以任何目的进行传播，违规者

造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁传播，违者自负法律责任！

目　录

单元 1　Docker 容器的安装　1

学习目标　1
学习情境　2
任务 1.1　容器技术的调研　2
任务描述　2
知识学习　2
任务实施　3
项目实训　8
任务 1.2　Docker 的安装　8
任务描述　8
知识学习　8
任务实施　10
项目实训　15
单元小结　15

单元 2　Docker 的使用和管理　17

学习目标　17
学习情境　18
任务 2.1　Docker 镜像管理　18
任务描述　18
知识学习　18
任务实施　19
项目实训　26
任务 2.2　Docker 镜像定制　26
任务描述　26
知识学习　26
任务实施　33
项目实训　38
任务 2.3　Docker 容器管理　38
任务描述　38
知识学习　38
任务实施　41
项目实训　47
单元小结　47

单元 3　Docker 仓库 Registry　49

学习目标　49
学习情境　50
任务 3.1　Docker 仓库与加速的配置和使用　50
任务描述　50
知识学习　50
任务实施　52
项目实训　53
任务 3.2　Docker 私有仓库与管理　54
任务描述　54
知识学习　54
任务实施　56
项目实训　58
单元小结　58

单元 4　Docker 网络、存储和接口　59

学习目标　59
学习情境　60
任务 4.1　Docker 存储　60
任务描述　60
知识学习　60
任务实施　60

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

项目实训 62
任务 4.2 Docker 网络 63
任务描述 63
知识学习 63
任务实施 65
项目实训 75
任务 4.3 Docker API 的使用 75
任务描述 75
知识学习 76
任务实施 76
项目实训 81
任务 4.4 Docker 底层技术应用 82
任务描述 82
知识学习 82
任务实施 90
项目实训 94
单元小结 95

单元 5 Docker 容器编排与集群 97

学习目标 97
学习情境 98
任务 5.1 Docker-compose 容器编排的使用 98
任务描述 98
知识学习 98
任务实施 99
项目实训 105
任务 5.2 Docker Swarm 集群的配置与管理 105
任务描述 105
知识学习 105
任务实施 106
项目实训 117
任务 5.3 Docker Consul 构建集群服务 117
任务描述 117
知识学习 117
任务实施 118
项目实训 121
单元小结 121

单元 6 容器服务管理平台 Rancher 123

学习目标 123
学习情境 124
任务 6.1 Rancher 安装、部署、使用 124
任务描述 124
知识学习 124
任务实施 125
项目实训 132
任务 6.2 基于 Rancher 构建持续集成 132
任务描述 132
知识学习 133
任务实施 135
项目实训 151
单元小结 152

单元 7 使用 Docker 构建持续集成 153

学习目标 153
学习情境 154
任务 7.1 构建 Nginx Web 应用 154
任务描述 154
知识学习 154
任务实施 155
项目实训 159
任务 7.2 持续集成管理软件 Jenkins 应用 159
任务描述 159
知识学习 159
任务实施 160
项目实训 161
任务 7.3 构建 Java Web 应用 162
任务描述 162
知识学习 162
任务实施 163
项目实训 167
任务 7.4 构建 Docker 负载均衡 167
任务描述 167
知识学习 167
任务实施 169
项目实训 177
单元小结 177

参考文献 178

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

单元1 Docker容器的安装

学习目标

知识目标

- 掌握容器技术的发展历史。
- 掌握容器技术的产生原因。
- 掌握容器技术的架构和优势。

技能目标

- 掌握 Docker 容器平台的安装条件。
- 掌握 Docker 容器平台在 CentOS 7 上的安装方法和步骤。
- 掌握 Docker 容器平台在 Windows 上的安装方法和步骤。
- 掌握 Docker 容器的安装验证与版本查看方法。
- 掌握简单的 Docker 启动方法。
- 掌握技术调研报告研究和编写方法。
- 掌握 Linux 基本知识的探索式学习方法。
- 拓展在各种系统和平台上安装 Docker 容器的能力。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

笔 记

学习情境

某公司已经实现了研发和服务运维的整合，公司的 OpenStack 云产品运行良好，但是不断增加的云端应用需求增加了硬件资源消耗，给公司带来了难题，公司已在云平台上运行了多台云主机，并消耗了大量的硬件资源。如何能够高效地利用硬件资源实现云服务呢？公司研发部了解到，Docker 容器技术可以提高硬件资源利用率，有效实现云服务，并能够提高研发部门的研发效率。于是决定采用 Docker 技术构建容器服务和研发运维持续集成环境，并安排工程师小魏对 Docker 技术进行调研分析和安装测试。

任务 1.1　容器技术的调研

任务描述

1. 了解 Docker 容器技术实现原理。
2. 了解所使用 Docker 技术的领域。
3. 通过 Docker 的优缺点，深入了解 Docker 容器技术。

知识学习

1. 容器技术的起源和实现原理

Docker 最初是 DotCloud 公司创始人 Solomon Hykes 在法国期间发起的一个公司内部项目。它是基于 DotCloud 公司多年云服务技术的一次革新，并于2013 年 3 月以 Apache 2.0 授权协议开源，主要项目代码在GitHub上进行维护。Docker 项目后来还加入了 Linux 基金会，并成立推动开放容器联盟。

Docker 自开源后便受到广泛的关注和讨论，至今其 GitHub 项目已经超过 36 000 个星标（Stat）和 10 000 多个分支（Fork）。甚至由于 Docker 项目的火爆，在 2013 年底，DotCloud 公司决定改名为 Docker。Docker 最初是在 Ubuntu 12.04 上开发实现的；Red Hat 则从 RHEL 6.5 开始对 Docker 进行支持；Google 也在其 PaaS 产品中广泛应用 Docker。

Docker 使用 Google 公司推出的Go 语言进行开发实现，基于 Linux 内核的Cgroups、NameSpace，以及AUFS类的Union FS等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其他隔离的进程，因此也称其为容器。最初的实现是基于LXC的；从 Docker 0.7 以后开始去除 LXC，转而使用自行开发的 Libcontainer；从 Docker 1.11 开始，则进一步演进为使用RunC和Containerd。

对于 Docker，目前的定义是一个开源的容器引擎，可以方便地对容器进行管理。其可以对镜像打包封装，以及引入 Docker Regsitry 对镜像进行统一的管理，构建了方便、快捷的“Build、Ship and Run”流程。它可以统一整个开发、测试和部署的环境和流程，极大地减少运维成本。另外，得益于容器技术带来的轻量级虚拟化，以及 Docker 在分层镜像应用上的创新，Docker 在磁盘占用、性能和效率方面相较于传统的虚拟化都有非常明显的提高，所以理所当然地 Docker 开始不断蚕食传统虚拟化的市场，具体的 Docker 版本变

非卖品，仅供非商业用途或交流学习使用

更见表 1-1。

表 1-1　Docker 版本变更

时　间	版　本	更 新 内 容
2013.7.18	v0.5	主要增加对外置存储、高级网络、自组织注册项的支持
2015.8.26	v1.8	更新 Toolbox 及编排工具
2015.10.20	v1.9.0	使用 Linux 容器引擎
2016.7.28	v1.12.0	改进了服务的负载均衡参数
2016.12.14	v1.12.4	更新容器文件系统

2. 容器和虚拟机的差异

Docker 和传统虚拟化方式有一些不同之处，传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整的操作系统，在该系统上再运行所需应用进程；而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，并且也没有进行硬件虚拟。因此，容器要比传统虚拟机更为轻便，传统虚拟化架构和 Docker 架构如图 1-1-1 和图 1-1-2 所示。

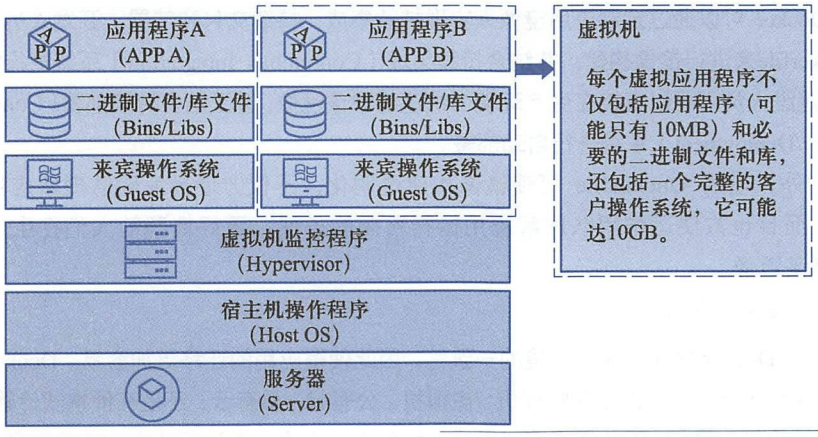


图 1-1-1
传统虚拟化架构

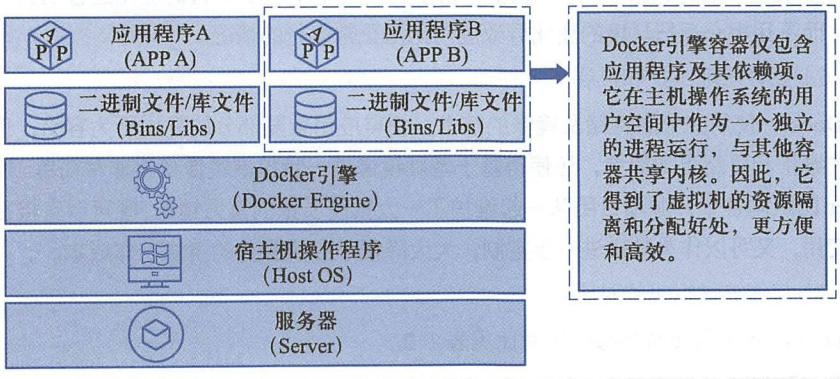


图 1-1-2
Docker 架构

任务实施

1. 探讨使用 Docker 的原因

作为一种新兴的虚拟化方式，Docker 跟传统的虚拟化方式相比具有众多的优势。

（1）更高效地利用系统资源

容器不需要进行硬件虚拟及运行完整操作系统等额外开销，且 Docker 对系统资源的

笔 记

利用率更高。无论是应用执行、文件存储，还是在减少内存损耗方面，都要比传统虚拟机技术更高效。因此，相比传统的虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。

（2）更快速的启动时间

传统的虚拟机技术启动应用服务往往需要数分钟，而 Docker 容器应用，由于直接运行于宿主内核，无须启动完整的操作系统，因此可以达到秒级甚至毫秒级的启动时间，大大节约了开发、测试、部署的时间。

（3）一致的运行环境

在开发过程中，一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 Bug 未在开发过程中被发现。而 Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境的一致性。

（4）持续交付和部署

对开发和运维人员来说，最希望的就是一次创建或配置可以在任意地方正常运行。使用 Docker 可以通过定制应用镜像来实现持续集成、持续交付和部署。开发人员可以通过Dockerfile来进行镜像构建，并结合持续集成（Continuous Integration）系统进行集成测试，而运维人员则可以直接在生产环境中快速部署该镜像，甚至结合持续部署（Continuous Delivery/Deployment）系统进行自动部署。

别外，使用 Dockerfile 可使镜像构建透明化，不仅方便开发团队理解应用运行环境，而且也方便运维团队理解应用运行所需的条件，更好地帮助人们在生产环境中部署该镜像。

（5）更轻松地迁移

由于 Docker 确保了执行环境的一致性，因此使得应用的迁移更加容易。Docker 可以在很多平台上运行，无论是在物理机、虚拟机、公有云、私有云，还是在便携式计算机中，其运行结果是一致的。因此用户可以很轻易地将在一个平台上运行的应用迁移到另一个平台上，而不用担心运行环境的变化导致应用无法正常运行的情况。

（6）更轻松地维护和扩展

Docker 使用的分层存储及镜像的技术，使得应用重复部分的复用更为容易，也使得应用的维护和更新更加简单，还使得基于基础镜像进一步扩展镜像变得非常简单。此外，Docker 团队同各个开源项目团队一起维护了一大批高质量的官方镜像，既可以直接在生产环境使用，又可以作为基础进一步定制，大大降低了应用服务的镜像制作成本。

（7）与传统虚拟机的对比

Docker 容器与传统虚拟机的对比见表 1-2。

表 1-2　Docker 容器与传统虚拟机对比

特　性	容　器	虚 拟 机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

笔 记

2. 调查使用 Docker 的情况

(1) Google 与容器

Google 从 2004 年起就已经开始使用容器技术了，并于 2006 年发布了 Cgroups 和 Imctfy 项目。Imctfy 是 Google 开源版本的容器栈，它提供了用来代替 LXC 的 Linux 应用容器。

Google 云平台的高级软件工程师 Joe Beda 于 2014 年在 Gluecon 上做了一个关于 Google 如何使用 Linux 容器技术的报告。报告中声称现在 Google 所有的应用都是运行在容器中的。Google 每周要启动超过 20 亿个容器，每秒钟要启动超过 3 000 个容器，这还不包括那些长期运行的容器。Google 同样也正在将容器集成到 Google 云平台中。

Google 于 2014 年推出了开源容器集群管理系统——Kubernetes，Kubernetes 就构建在 Docker 之上。

(2) 京东与容器

2013 年，京东弹性云从 KVM 起步。京东弹性云首先应用于京东私有云上。在选择 Docker 之前，尝试了 KVM 虚拟化方案，各方反馈性能不够好，无法满足大促的需求。

2014 年 10 月，京东开始思考用 Docker 重构，在 2014 年底，决定采用 Docker 容器化方案，通过基于 Docker 镜像的方式进行发布，其快速的弹性伸缩能力适合京东的大规模生产需求，并且其具备轻量、高性能和便捷性的优势。京东首先使用图片系统来验证其性能和弹性伸缩能力，之后再逐步推广到如单品页这样的核心系统，并进行网络的优化，以满足京东应用的性能，京东容器架构如图 1-1-3 所示。

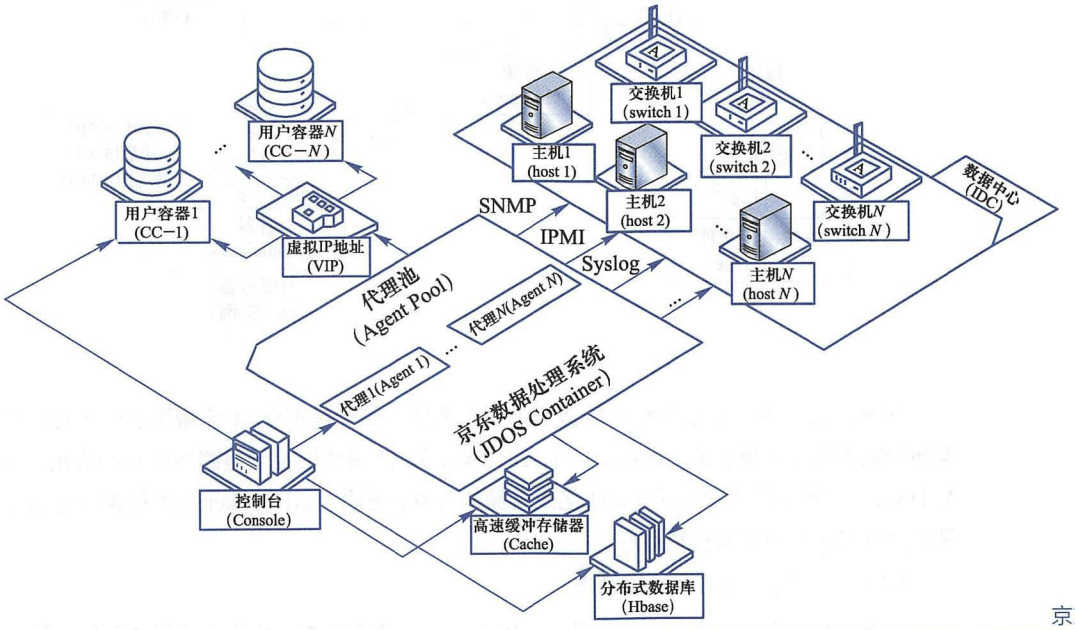


图 1-1-3
京东容器架构

在 2015 年的 618 购物节中，京东启用了基于 Docker 的容器技术来承载大促的关键业务（图片展现、单品页、团购页），当时在基于 Docker 容器的弹性云项目中已经有近万个 Docker 容器在线上环境运行，并且经受了大流量的考验。在 2016 年的 618 购物节中，弹性云项目更是担当重任，全部应用系统和大部分的 DB（数据库）服务都落在 Docker

上。在经受了 2015 年的 618 购物节和双 11 狂欢节大流量的考验后，2016 年的 618 购物节和双 11 狂欢节的数据中心全部采用弹性云进行建设，并把 2016 年 618 购物节的所有流量全部落到弹性云上进行处理。

（3）阿里与容器

T4 是阿里在 2011 年的时候基于 LinuxContainer（LXC）开发的容器技术基础设施。T4 是从阿里内部的资源管理和日常运维中土生土长出来的产品，在诞生的第一天就针对内部基础设施、运维工具甚至运维习惯做了很多特别的设计。

阿里从 2015 年开始对 Docker 和 T4 做了一些修改整合后，将两者融合为了一个产品，相当于既让 T4 具备了 Docker 的镜像能力，又让 Docker 具备了 T4 具有的对内部运维体系的友好性，并且能够运行在内部的 AliOS5u7 和 2.6.32 内核上。这个产品在阿里内部称为 AliDocker，如图 1-1-4 所示。

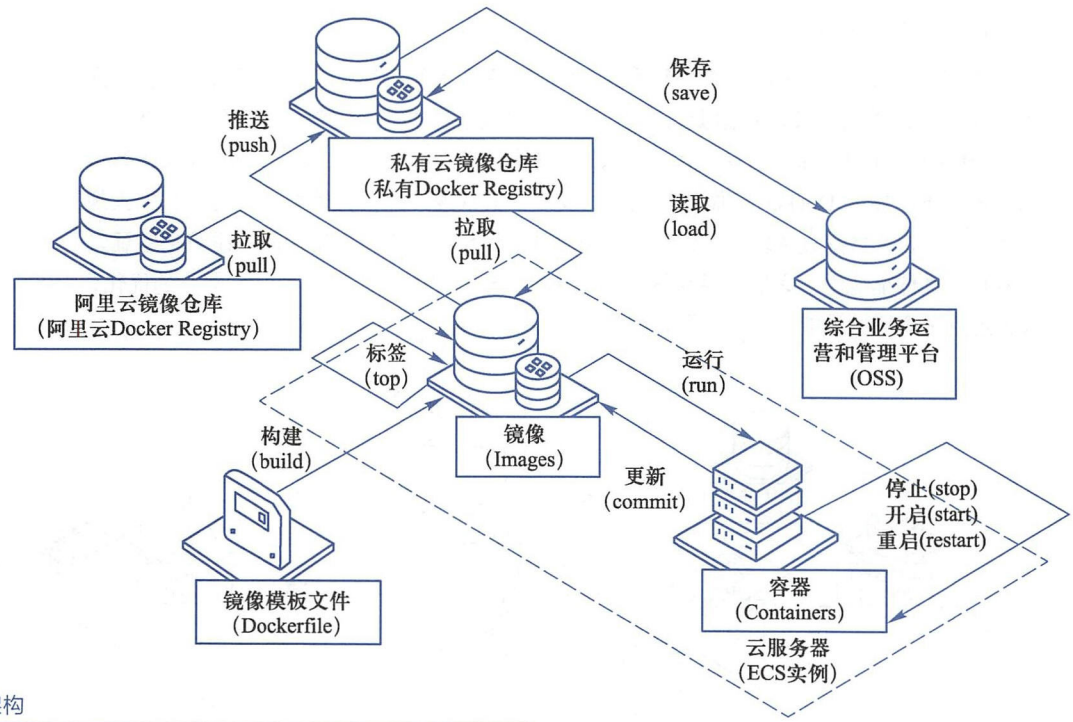


图 1-1-4
阿里云容器架构

阿里成立了专门的项目组来快速推进这个事情，目标是把双 11 流量覆盖的核心应用全部升级为镜像化模式的 Docker 应用。2016 年，双 11 最终交易全链路所有核心应用，在 AliDocker 容器中圆满交出了 1 207 亿成交额的答卷，证明了 AliDocker 技术体系在电商交易的大规模应用中能够担负重任。

（4）全球用户 2016 年调研情况

根据 CloudFoudry 的 2016 年用户分析报告，目前容器主要用于继续集成开发、微服务资源分享、运行环境的版本控制、操作系统和替换虚拟机，如图 1-1-5 所示。

主要使用的容器集群管理架构服务的调查报告表明，当前主流的管理服务依然是 Docker Swarm、Google Kubernetes、Amazon ECS，如图 1-1-6 所示。其他平台包括 Apache MESOS、Rancher 的 CATTLE 等，只占据使用率的 21%。

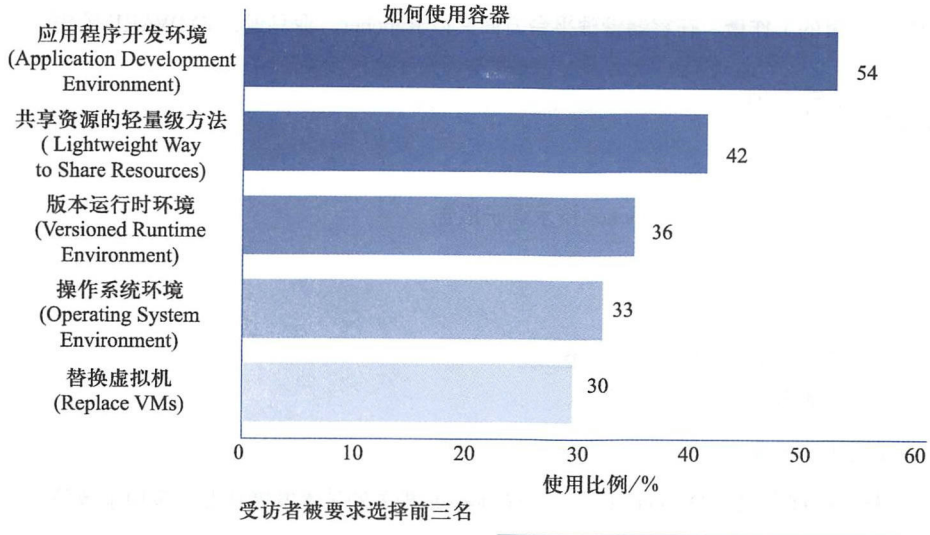


图 1-1-5
容器主要使用场景

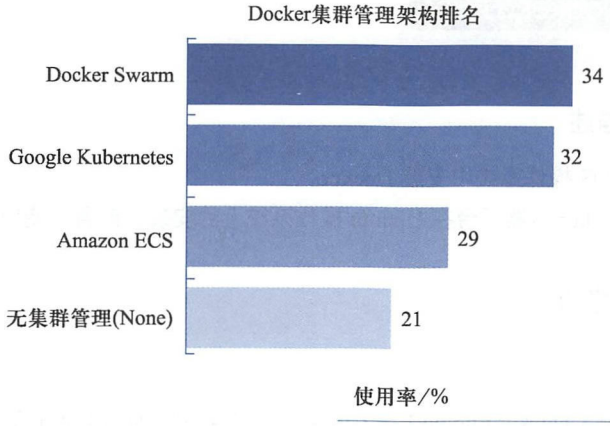


图 1-1-6
Docker 集群管理架构
服务

前面介绍的是容器在互联网公司生产系统的应用。在生产环境中，Docker 容器还有一个重要的应用场景，就是开发的持续集成，如图 1-1-7 所示。

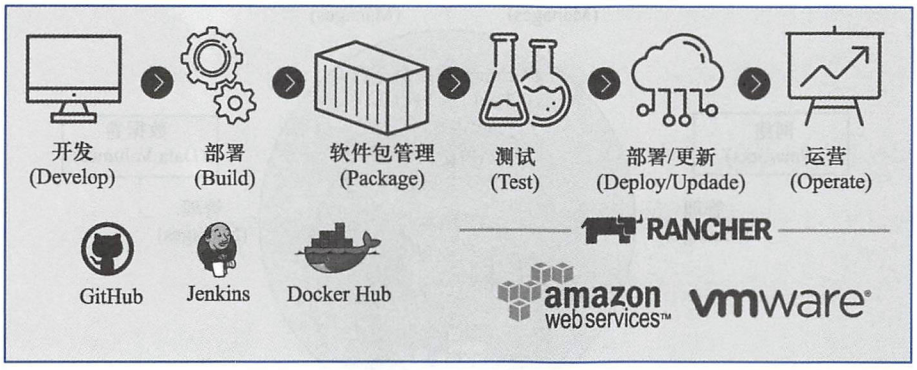


图 1-1-7
持续集成

在常见的一些企业和应用中，Docker 都应用得非常广泛，例如，GitHub、Jenkins 和 Docker Hub 基本使用 Docker 作为其底层的系统，为软件的部署、构建、安装包管理、测试、使用更新和运行提供底层的环境支撑，可见，容器为开发、测试、预生产和生产环境的自动部署提供整体可视化的主机、容器、网络及存储管理，大幅简化运维人员故障排除

和生产部署的工作量。在容器管理平台方面又以 Rancher、亚马逊、VMWARE 领先。

笔 记

项目实训

【实训题目】

查询相关知识后编写 Docker 技术调研报告

【实训目的】

1. 了解 Docker 容器。
2. 了解容器和虚拟化技术的区别。
3. 了解容器应用的技术领域。

【实训内容】

从网络搜索相关 Docker 知识，了解 Docker 相关的技术发展状况和应用领域等。

任务 1.2　Docker 的安装

任务描述

1. 在 CentOS 操作系统中安装 Docker。
2. 实现 Docker 容器平台在 Ubuntu 操作系统上的安装、部署、使用。

知识学习

1. Docker 引擎

Docker 引擎（Docker Engine）是 C/S 架构，其架构图如图 1-2-1 所示。

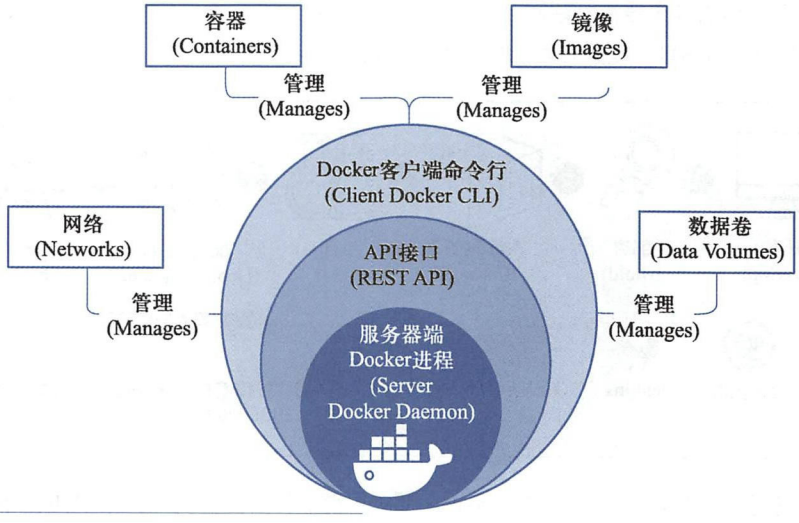


图 1-2-1
Docker 服务的架构图

它主要由以下几个部件组成。

（1）服务器（Docker Daemon）

后台运行的是 Docker Daemon 进程。Daemon 进程用于管理 Docker 对象，包括镜像

（Images）、容器（Containers）、网络（Networks）和数据卷（Data Volumes）。

（2）REST 接口

该接口是同 Daemon 交互的 API 接口。

（3）客户端（Docker Client）

客户端使用 REST API 接口同 Docker Daemon 进行访问。

2. Docker 平台组成

一个完整的 Docker 服务包括服务器、客户端、镜像、仓库和容器，如图 1-2-2 所示。

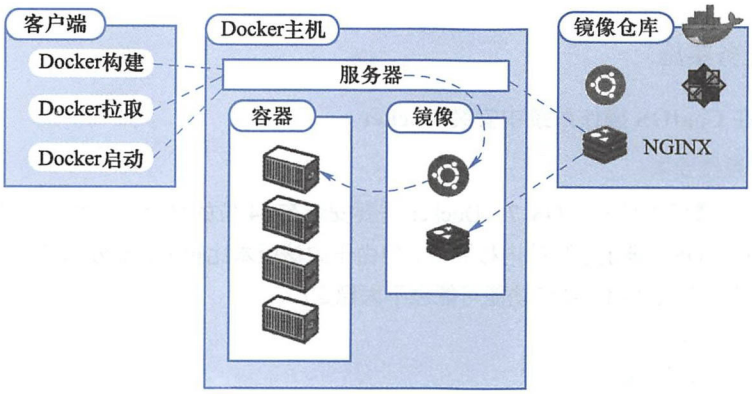


图 1-2-2
Docker 服务组成图

（1）镜像

镜像是一个只读模板，用于创建容器，由 Dockerfile 文本描述镜像的内容。镜像定义类似面对对象的类，从一个基础镜像（Base Image）开始。构建一个镜像实际就是安装、配置和运行的过程。Docker 镜像基于 UnionFS 把以上过程进行分层（Layer）存储，这样更新镜像可以只更新变化的层。Docker 的描述文件为 Dockerfile，Dockerfile 是一个文本文件，基本指令包括以下内容。

① FROM：定义基础镜像。

② MAINTAINER：作者或维护者。

③ RUN：运行 Linux 命令。

④ ADD：增加文件或目录。

⑤ EVN：定义环境变量。

⑥ CMD：运行进程。

（2）容器

容器是一个镜像的运行实例。容器由镜像创建，运行过程。

运行 Ubuntu 操作系统镜像，-i 前台交互模型，运行命令为/bin/bash，代码如下：

```
#  docker run -i -t ubuntu /bin/bash
```

运行过程如下：

① 拉取（Pull）镜像，Docker Engine 检查 Ubuntu 镜像是否存在。如果本地已经存在，则使用该镜像创建容器；如果不存在，则 Docker Engine 从镜像库拉镜像。

② 使用该镜像创建新容器。

③ 分配文件系统，挂载一个读写层，在读写层加载镜像。

④ 分配网络/网桥接口，创建一个网络接口，让容器和主机通信。

⑤ 从可用的 IP 池选择 IP 地址，分配给容器。

⑥ 执行命令/bin/bash。

⑦ 捕获和提供执行结果。

（3）仓库

Docker 仓库是 Docker 镜像库。Docker Registry 也是一个容器。Docker Hub 是 Docker 公司提供的互联网公共镜像仓库。用户可以构建自己本地的镜像仓库，国内有些公司也构建了镜像仓库，这部分内容在后面的章节会有详细说明。

任务实施

1. 在 CentOS 操作系统中安装 Docker

（1）系统要求

Docker 最低支持 CentOS 7，Docker 需要安装在 64 位的平台中，并且内核版本不低于 3.10。CentOS 7 满足最低的内核要求，但由于内核版本比较低，部分功能（如 overlay 2 存储层驱动）无法使用，部分功能可能也不太稳定。

（2）手动安装

① 添加内核参数。

默认配置下，在 CentOS 中使用 Docker 可能会遇到下面的这些警告信息：

```
WARNING: bridge-nf-call-iptables is disabled
WARNING: bridge-nf-call-ip6tables is disabled
```

添加内核配置参数可启用这些功能，并可打开内核转发功能。

编辑配置文件/etc/sysctl.conf，添加以下内容。

```
# vi /etc/sysctl.conf
net.ipv4.ip_forward = 1
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
```

② 重新加载 sysctl.conf。

```
# sysctl -p
net.ipv4.ip_forward = 1
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
```

③ 删除 iptables 防火墙规则。

```
# iptables –F
# iptables –X
# iptables –Z
# /usr/sbin/iptables-save
```

（3）添加 yum 源

虽然 CentOS 的软件源 Extras 中有 Docker 安装镜像，但是不建议使用系统源中的这

个版本，它的版本相对比较旧，而且并非 Docker 官方维护的版本。因此，需要使用 Docker 官方提供的 CentOS 软件源，在提供的软件包中也包含了这个安装包，只需要配置好 yum 仓库源即可正常安装使用。

① 执行下面的命令以添加 yum 软件源。

```
# cat /etc/yum.repos.d/yum.repo
[CentOS]
name=CentOS
baseurl=ftp://10.0.0.254/file/cr/2017/CentOS7.2/
gpgcheck=0
enabled=1
[iaas]
name=iaas
baseurl=ftp://10.0.0.254/file/cr/2017/iaas/iaas-repo/
gpgcheck=0
enabled=1
```

注意

这个 yum 源可以根据自己的实验环境来修改。

② 配置 DNS 服务器地址。

```
# vi /etc/resolv.conf
; generated by /usr/sbin/dhclient-script
search openstacklocal
nameserver 114.114.114.114
nameserver 223.5.5.5
```

对所有节点配置文件/etc/sysconfig/docker 修改如下配置。

```
ADD_REGISTRY='--add-registry 10.0.6.83:5000'
INSECURE_REGISTRY='--insecure-registry 10.0.6.83:5000'
```

IP 地址是仓库地址。

③ 配置镜像加速器。

国内访问 Docker Hub 有时会遇到困难，此时可以配置镜像加速器。国内很多云服务商都提供了加速器服务，例如阿里云加速器、DaoCloud 加速器、灵雀云加速器。此例中使用的是DaoCloud 加速器。

```
# curl -sSL  https://get.daocloud.io/daotools/set_mirror.sh | sh -s
http://ef0cb1d0.m.daocloud.io
Success.
You need to restart docker to take effect: sudo systemctl restart docker
# systemctl restart docker
```

（4）安装 Docker

① 检查内核。

```
# uname -a
Linux server 3.10.0-229.el7.x86_64 #1 SMP Fri Mar 6 11:36:42 UTC 2015
x86_64 x86_64 x86_64 GNU/Linux
```

② 检查 Device Mapper（存储驱动）。

```
# ls -l /sys/class/misc/device-mapper
ls: cannot access /sys/class/misc/device-mapper: No such file or directory
# grep device-mapper /proc/devices
```

③ 以上检查说明没有安装 Device Mapper，需要安装 Device Mapper 软件包，代码如下：

```
# sudo yum install -y device-mapper
```

④ 加载 Device Mapper 模块。

```
# modprobe dm-mod
```

⑤ 验证 Device Mapper 安装。

```
# ls -l /sys/class/misc/device-mapper
lrwxrwxrwx 1 root root 0 Dec 23 09:39 /sys/class/misc/device-mapper
-> ../../devices/virtual/misc/device-mapper
```

⑥ 所有节点更新 yum 源。

```
# yum update
```

⑦ 安装 Docker-io。

```
# yum -y install docker-io
Loaded plugins: fastestmirror
base
| 3.6 kB   00:00:00
CentOS
| 3.6 kB   00:00:00
extras
| 3.4 kB   00:00:00
iaas
| 2.9 kB   00:00:00
updates
| 3.4 kB   00:00:00
（1/4）: base/7/x86_64/group_gz
| 155 kB   00:00:01
（2/4）: extras/7/x86_64/primary_db
| 183 kB   00:00:02
（3/4）: updates/7/x86_64/primary_db
| 1.2 MB   00:00:02
（4/4）: base/7/x86_64/primary_db
…
Dependency Updated:
```

版权所有，严禁传播，违者自负法律责任！


```
  docker-common.x86_64 2:1.10.3-59.el7.CentOS
libgudev1.x86_64 0:219-30.el7_3.6
  libselinux.x86_64 0:2.5-6.el7
libselinux-python.x86_64 0:2.5-6.el7
  libselinux-utils.x86_64 0:2.5-6.el7
libsemanage.x86_64 0:2.5-4.el7
  libsemanage-python.x86_64 0:2.5-4.el7
libsepol.x86_64 0:2.5-6.el7
  oci-systemd-hook.x86_64 1:0.1.4-6.git337078c.el7
policycoreutils.x86_64 0:2.5-9.el7
  policycoreutils-python.x86_64 0:2.5-9.el7
selinux-policy.noarch 0:3.13.1-102.el7_3.7
  selinux-policy-targeted.noarch 0:3.13.1-102.el7_3.7
setools-libs.x86_64 0:3.3.8-1.1.el7
  systemd-libs.x86_64 0:219-30.el7_3.6
systemd-sysv.x86_64 0:219-30.el7_3.6
Complete!
```

⑧ 启动服务器。

```
# service docker start
Redirecting to /bin/systemctl start  docker.service
# /bin/systemctl start  docker.service
```

⑨ 开机启动 Docker。

```
# systemctl enable docker.service
```

⑩ 检查 Docker 是否正确安装。

```
# docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 1.10.3
Storage Driver: devicemapper
 Pool Name: docker-253:1-260046984-pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: xfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 11.8 MB
 Data Space Total: 107.4 GB
 Data Space Available: 106.3 GB
 Metadata Space Used: 581.6 kB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.147 GB
```

版权所有，严禁传播，违者自负法律责任！

2. 在 Windows 中安装 Docker Toolbox

Docker Toolbox 是一个 Docker 组件的集合，包括一个极小的虚拟机。用户可通过 Docker Toolbox 在 PC（Windows/ OS X）构建一个入门的 Docker 环境。

首先需要从网上下载安装文件 Docker Toolbox.setup。打开后进入安装界面，如图 1-2-3 所示，然后单击 Next 按钮进行一键安装。

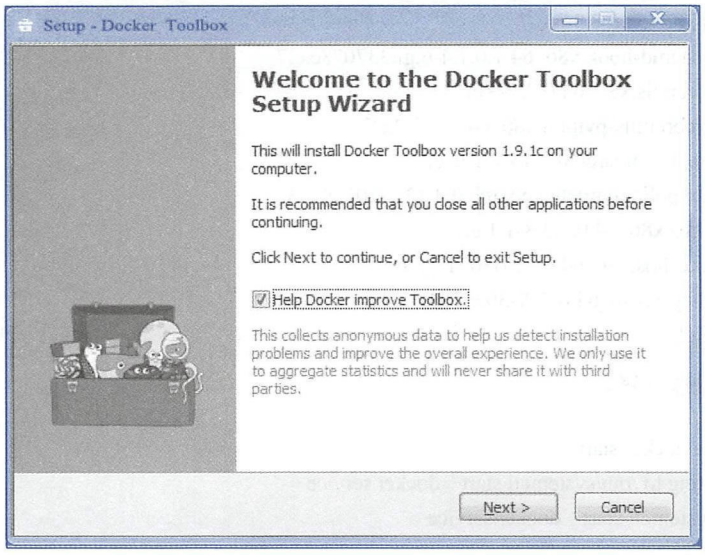


图 1-2-3
安装 Docker Toolbox 的界面

安装完成会出现如图 1-2-4 所示的界面。

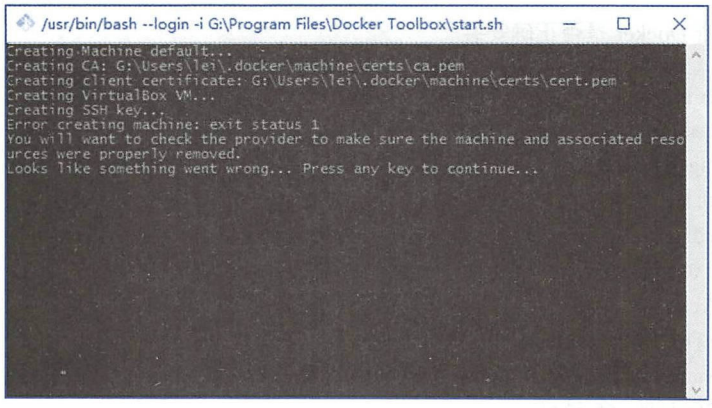


图 1-2-4
Docker Toolbox 安装完成界面

此时可以看到桌面上生成了 3 个快捷方式图标，如图 1-2-5 所示。

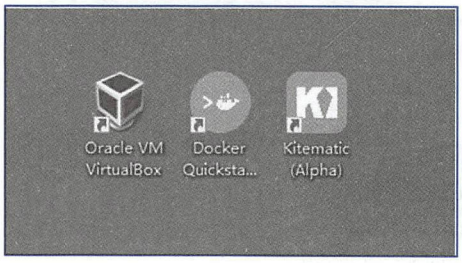


图 1-2-5
安装成功后的桌面快捷方式图标

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

如果由于 Windows 操作系统的版本问题不能正常使用 VirtualBox 创建虚拟机，则需要替换主题包，具体操作如下。

① 下载还原软件，这里用的是 UniversalThemePatcher-x64。

② 下载主题包文件：uxtheme.dll、themeui.dll、themeservice.dll。

③ 将这个主题包文件复制到 Windows 系统的/Window/System32/中（主题包文件被系统占用，无法替换，复制过程可在 PE 系统中进行）。

④ 以管理员身份运行 UniversalThemePatcher-x64，破解后还原，还原之后需要重启。

项目实训

【实训题目】

完成 Ubuntu 操作系统下的 Docker 安装。

【实训目的】

1. 掌握 Docker 容器平台的安装
2. 掌握 Ubuntu 操作系统的安装
3. 掌握 Docker 在 Ubuntu 系统中的安装

【实训内容】

1. 安装 Ubuntu 操作系统。
2. 在已安装的 Ubuntu 系统中安装 Docker 容器平台。

单元小结

本单元介绍了 Docker 容器的原理，以及 Docker 容器技术如此之火的原因。通过和传统的虚拟机对比，用户明确了 Docker 技术与传统技术的优缺点。Docker 技术之所以被各个领域广泛使用，也正是因为它本身的这些优点。通过对 Docker 环境的搭建，用户不仅掌握了 Docker 环境的部署，更深入地了解了 Docker 技术的特性与优越性。通过本单元的学习，用户可以产生对 Docker 容器技术的学习兴趣。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

单元 2 Docker 的使用和管理

学习目标

知识目标

- 了解镜像的基本概念。
- 掌握镜像的使用方法。
- 了解容器的基本概念。
- 掌握容器的管理方法。

技能目标

- 学会镜像的简单运维。
- 掌握容器的基本操作。
- 掌握容器的运维管理。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

笔 记

学习情境

公司研发部根据工程师小魏对 Docker 的技术分析和对 Docker 容器的测试，得出了使用 Docker 容器是轻量级虚拟化方案，可以在研发部内推广该技术。工程师小魏负责对容器技术的基本使用做出一套规范性的手册以供团队人员学习，手册上主要包括镜像的使用管理、容器的运行及维护。

任务 2.1　Docker 镜像管理

任务描述

1. 了解 Docker 镜像的本地存储。
2. 使用 Docker 命令实现镜像的线上查找、获取。
3. 使用 Docker 命令实现镜像的线下运行、列出、删除。

知识学习

1. Docker 镜像的定义

从整体的角度来讲，一个完整的 Docker 镜像可以支撑一个 Docker 容器的运行，在 Docker 容器运行过程中主要提供文件系统视角。例如一个 ubuntu:14.04 的镜像，提供了一个基本的 ubuntu:14.04 的发行版，当然此镜像是不包含操作系统 Linux 内核的。

说到此，就需要注意一下 Linux 内核和 ubuntu:14.04Docker 镜像的区别了。传统虚拟机安装 ubuntu:14.04 会包含两部分：某一个 Linux 内核的发行版本（如 Linux 3.8 版本的内核）和一个特定的 Ubuntu 发行版（这部分内容不包含 Linux 内核，但是包含 Linux 之外的软件管理方式和软件驱动，如 apt-get 软件管理包等）。

理解以上内容之后就可以理解，为什么在一个 Linux 内核版本为 3.8 的 ubuntu:14.04 基础上可以把 Linux 内核版本升级到 3.18，而 Ubuntu 的版本依然是 14.04。最主要的就是：Linux 内核版本与 Ubuntu 操作系统发行版之间的区别。

Linux 内核与 Ubuntu 操作系统发行版组成一台工作的机器让用户体验。那么灵活替换 Ubuntu 操作系统发行版是不是也可以实现呢？其实 Docker 很方便地利用了这一点，技术手段就是 Docker 镜像。

在 Docker 的架构中，Docker 镜像类似于“Ubuntu 操作系统发行版”，可以在任何满足要求的 Linux 内核之上运行。简单一点来说，有“Debian 操作系统发行版”Docker 镜像、“Ubuntu 操作系统发行版”Docker 镜像。如果在 Debian 镜像中安装 MySQL 5.6，那么可以将其命名为 mysql:5.6 镜像；如果在 Debian 镜像中安装 Golang 1.3，那么可以将其命名为 golang:1.3 镜像。以此类推，用户可以根据自己安装的软件得到任何自己想要的镜像。

那么镜像最后的作用是什么呢？很好理解，回到 Linux 内核上来运行，通过镜像来运行时，人们常常将提供的环境称为容器。

以上内容是从宏观的角度来介绍 Docker 镜像是什么，下面再从微观的角度进一步深

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

入介绍 Docker 镜像。刚才提到了，在 Debian 镜像中安装 MySQL 5.6，就成了 mysql:5.6 镜像，其实此时 Docker 镜像的层级概念就体现出来了。底层为一个 Debian 操作系统镜像，上面叠加一个 mysql 层，就完成了一个 mysql 镜像的构建。这样层级概念就不难理解了，此时，一般 Debian 操作系统镜像称为 mysql 镜像层的父镜像。

层级管理的方式大大便捷了 Docker 镜像的分发与存储。说到分发，人们自然会联想到 Docker 镜像的灵活性、传输的便捷性，以及高超的移植性。Docker Hub，作为全球的镜像仓库，作为 Docker 生态中的数据仓库，将全世界的 Docker 数据汇聚在一起，是 Docker 生态的命脉。

Docker 的两方面的技术非常重要，第一是 Linux 容器方面的技术，第二是 Docker 镜像的技术。从技术本身来讲，两者的可复制性很强，不存在绝对的技术难点，然而由于 Docker Hub 存在大量数据的原因，导致 Docker Hub 的可复制性几乎不存在，这需要一个生态的营造。

2. Docker 镜像的内容

前面介绍了 Docker 镜像是什么，接下来看看 Docker 镜像中有哪些内容?

介绍之前，先分享一下作者在接触 Docker 的两年时间中对 Docker 镜像内容认识的变化。

- 第一阶段：初步接触 Docker。相信很多爱好者都会有这样一个认识：Docker 镜像代表一个容器的文件系统内容。
- 第二阶段：初步接触联合文件系统。联合文件系统的概念让人们意识到镜像层级管理的技术，每一层镜像都是容器文件系统内容的一部分。
- 第三阶段：研究镜像与容器的关系。容器是一个动态的环境，每一层镜像中的文件都属于静态内容，然而 Dockerfile 中的 ENV、VOLUME、CMD 等内容最终都需要落实到容器的运行环境中，而这些内容均不可能直接坐落到每一层镜像所包含的文件系统内容中，此时每一个 Docker 镜像还会包含 json 文件记录与容器之间的关系。

因此，Docker 镜像的内容主要包含两个部分：第一，镜像层文件内容；第二，镜像 json 文件。

任务实施

1. 获取镜像

Docker Hub 上有大量高质量的镜像可以用，从 Docker Registry 获取镜像的命令是 docker pull。其命令格式为：

```
docker pull [选项] [Docker Registry 地址]<仓库名>:<标签>
```

具体的选项可以通过 docker pull --help 命令看到。

（1）从 Docker Hub 拉取 Ubuntu 镜像

```
# docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu
bf5d46315322: Pull complete
9f13e0ac480c: Pull complete
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
e8988b5b3097: Pull complete
40af181810e7: Pull complete
e6f7c7e5c03e: Pull complete
Digest: sha256:147913621d9cdea08853f6ba9116c2e27a3ceffecf3b492983ae97c3d643fbbe
Status: Downloaded newer image for ubuntu:14.04
```

（2）从私有库拉取 Ubuntu 镜像

```
# docker pull 192.168.200.11:5000/ubuntu/14.04.3.tar:latest
Trying to pull repository 192.168.200.11:5000/ubuntu/14.04.3.tar ...
latest: Pulling from 192.168.200.11:5000/ubuntu/14.04.3.tar
```

（3）查看 Ubuntu 镜像是否下载到了本地

```
# docker images
REPOSITORY                                    TAG         IMAGE ID        CREATED         SIZE
192.168.200.11:5000/rancher/agent-instance    v0.8.3.tar  6dccaa66a75d    6 months ago    330.9 MB
192.168.200.11:5000/rancher/agent             v1.0.2.tar  deb5c6987c83    6 months ago    454.3 MB
192.168.200.11:5000/ubuntu/14.04.3.tar        latest      ebdc8e295a2e    11 months ago   187.9 MB
```

注意

本地镜像都保存在 Docker 宿主机的/var/lib/docker 目录下。

```
# cd /var/lib/docker/
# ll
total 8
drwx------. 15 root root 4096 Dec 24 19:58 containers
drwx------.  5 root root   50 Dec 23 14:47 devicemapper
drwx------.  3 root root   25 Dec 23 13:02 image
drwxr-x---.  3 root root   18 Dec 23 13:02 network
drwx------.  6 root root 4096 Dec 24 19:58 tmp
drwx------.  2 root root    6 Dec 23 13:02 trust
drwx------.  3 root root   77 Dec 23 14:58 volumes
```

2. 运行镜像

有了镜像后，就可以以这个镜像为基础启动一个容器来运行。以上面的 ubuntu:14.04 为例，如果打算启动 ubuntu 里面的 bash 且进行交互式操作的话，那么可以执行下面的命令。

```
# docker run -it --rm ubuntu:14.04 bash
root@e7009c6ce357:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="14.04.5 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.5 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
root@e7009c6ce357:/# exit

# docker run -t -i --name next_container ubuntu/14.04.3.tar:latest /bin/bash
root@917b059d9f30:/#
```

docker run 就是运行容器的命令，具体格式会在后面的章节进行讲解。

最后通过 exit 退出这个容器。

3. 列出镜像

（1）查询镜像

要想列出已经下载下来的镜像，可以使用 docker images 命令。

```
# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
redis               latest              5f515359c7f8        5 days ago          183 MB
nginx               latest              05a60462f8ba        5 days ago          181 MB
mongo               3.2                 fe9198c04d62        5 days ago          342 MB
<none>              <none>              00285df0df87        5 days ago          342 MB
ubuntu              16.04               f753707788c5        4 weeks ago         127 MB
ubuntu              latest              f753707788c5        4 weeks ago         127 MB
ubuntu              14.04               1e0c3dd64ccd        4 weeks ago         188 MB
```

列表包含了仓库名、标签、镜像 ID、创建时间及所占用的空间。

镜像 ID 是镜像的唯一标志，一个镜像可以对应多个标签。因此，在上面的例子中，可以看到 ubuntu:16.04 和 ubuntu:latest 拥有相同的 ID，因为它们对应的是同一个镜像。

（2）镜像体积

如果仔细观察就会注意到，这里，标志所占用的空间和在 Docker Hub 上看到的镜像大小不同。比如 ubuntu:16.04 的镜像大小，在这里是 127 MB，但是在Docker Hub显示的却是 50 MB。这是因为 Docker Hub 中显示的体积是压缩后的体积。在镜像下载和上传过程中，镜像是保持着压缩状态的，因此 Docker Hub 所显示的大小是网络传输中更关心的流量大小。而 docker images 显示的是镜像下载到本地后展开的大小，准确地说是展开后的各层所占空间的总和，因为镜像到本地后，查看空间的时候，更关心的是本地磁盘空间占用的大小。

另外一个需要注意的问题是，docker images 列表中的镜像体积总和并非是所有镜像的实际硬盘消耗。由于 Docker 镜像是多层存储结构，并且可以继承、复用，因此不同的镜像可能会因为使用相同的基础镜像，从而拥有共同的层。由于 Docker 使用 Union FS，相同的层只需要保存一份即可，因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小得多。

（3）虚悬镜像

在上面的镜像列表中还可以看到一个特殊的镜像，这个镜像既没有仓库名，也没有标签，均为“<none>”。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
<none>          <none>          00285df0df87        5 days ago          342 MB
```

这个镜像原本是有镜像名和标签的，原来为“mongo:3.2”，随着官方镜像维护，发布了新版本后，重新执行“docker pull mongo:3.2”命令时，“mongo:3.2”这个镜像名被转移到了新下载的镜像上，而旧的镜像上的这个名称则被取消，从而成为了“<none>”这个标签的镜像。除了 docker pull 可能导致这种情况，docker build 也同样可以导致这种现象。由于新旧镜像同名，旧镜像名称被取消，因而出现仓库名、标签均为“<none>”的镜像。这类无标签镜像也被称为虚悬镜像（dangling image），可以用下面的命令专门显示这类镜像：

```
# docker images -f dangling=true
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              00285df0df87        5 days ago          342 MB
```

一般来说，虚悬镜像已经失去了存在的价值，是可以随意删除的，可以用下面的命令删除。

```
# docker rmi $(docker images -q -f dangling=true)
```

（4）中间层镜像

为了加速镜像构建，重复利用资源，Docker 会利用中间层镜像。所以在使用一段时间后，可能会看到一些依赖的中间层镜像。默认的 docker images 列表中只会显示顶层镜像，如果希望显示包括中间层镜像在内的所有镜像，则需要加“-a”参数。

```
# docker images -a
```

这样会看到很多无标签的镜像，与之前的虚悬镜像不同，这些无标签的镜像很多都是中间层镜像，是其他镜像所依赖的镜像。这些无标签镜像不应该删除，否则会导致上层镜像因为依赖丢失而出错。实际上，这些镜像也没必要删除，因为之前说过，相同的层只会存一遍。当然只要删除那些依赖它们的镜像，这些依赖的中间层镜像也会被连带删除。

（5）列出部分镜像

在不加任何参数的情况下，docker images 会列出所有顶级镜像，但是有时候只希望列出部分镜像。docker images 有好几个参数可以帮助做到这个事情。

（6）根据仓库名列出镜像

```
# docker images ubuntu
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              16.04               f753707788c5        4 weeks ago         127 MB
ubuntu              latest              f753707788c5        4 weeks ago         127 MB
ubuntu              14.04               1e0c3dd64ccd        4 weeks ago         188 MB
```

（7）列出特定的某个镜像（即指定仓库名和标签）

```
$ docker images ubuntu:16.04
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              16.04               f753707788c5        4 weeks ago         127 MB
```

（8）以特定格式显示

默认情况下，docker images 会输出一个完整的表格，但是并非所有的时候都会需要

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

这些内容。例如删除虚悬镜像的时候，需要利用 docker images 把所有虚悬镜像的 ID 列出来，然后才可以使用 docker rmi 命令来删除指定的这些镜像，这个时候就用到了“-q”参数。

```
# docker images -q
5f515359c7f8
05a60462f8ba
fe9198c04d62
00285df0df87
f753707788c5
f753707788c5
1e0c3dd64ccd
```

例如，下面的命令会直接列出镜像结果，并且只包含镜像 ID 和仓库名：

```
# docker images --format "{{.ID}}: {{.Repository}}"
5f515359c7f8: redis
05a60462f8ba: nginx
fe9198c04d62: mongo
00285df0df87: <none>
f753707788c5: ubuntu
f753707788c5: ubuntu
1e0c3dd64ccd: ubuntu
```

也可以以表格等距显示，并且有标题行，和默认一样，不过需要自己定义列：

```
# docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"
IMAGE ID            REPOSITORY          TAG
5f515359c7f8        redis               latest
05a60462f8ba        nginx               latest
fe9198c04d62        mongo               3.2
00285df0df87        <none>              <none>
f753707788c5        ubuntu              16.04
f753707788c5        ubuntu              latest
1e0c3dd64ccd        ubuntu              14.04
```

4. 查找镜像

要通过 docker search 来查找镜像，首先要查找本地，然后是本地库，最后是 Docker Hub 库。

```
# docker search ubuntu
INDEX       NAME                                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
docker.io   docker.io/ubuntu                    Ubuntuis a Debian-based Linux operating s...    5280                 [OK]
docker.io   docker.io/ubuntu-upstart            Upstart is an event-based replacement for ...   69                   [OK]
docker.io   docker.io/rastasheep/ubuntu-sshd    Dockerized SSH service, built on top of of...   61                   [OK]
docker.io   docker.io/consol/ubuntu-xfce-vnc    Ubuntu container with "headless" VNC sessi...   34                   [OK]
```


非卖品，仅供非商业用途或交流学习使用

docker.io	docker.io/torusware/speedus-ubuntu	Always updated official Ubuntu dockerimag...	27	[OK]
docker.io	docker.io/ubuntu-debootstrap	debootstrap --variant=minbase --components...	27	[OK]
docker.io	docker.io/nickistre/ubuntu-lamp	LAMP server on Ubuntu	13	[OK]
docker.io	docker.io/nuagebec/ubuntu	Simple always updated Ubuntu docker images...	12	[OK]
docker.io	docker.io/nickistre/ubuntu-lamp-wordpress	LAMP on Ubuntu with wp-cli installed	8	[OK]
docker.io	docker.io/nimmis/ubuntu	This is a docker images different LTS vers...	6	[OK]
docker.io	docker.io/maxexcloo/ubuntu	Base image built on Ubuntu with init, Supe...	2	[OK]
docker.io	docker.io/admiringworm/ubuntu	Base Ubuntu images based on the official u...	1	[OK]
docker.io	docker.io/darksheer/ubuntu	Base Ubuntu Image -- Updated hourly	1	[OK]
docker.io	docker.io/jordi/ubuntu	Ubuntu Base Image	1	[OK]
docker.io	docker.io/teamrock/ubuntu	TeamRock's Ubuntu image configured with AW...	0	[OK]
docker.io	docker.io/ustclug/ubuntu	Ubuntu image for docker with USTC mirror	0	[OK]
docker.io	docker.io/vcatechnology/ubuntu	A Ubuntu image that is updated daily	0	[OK]
docker.io	docker.io/webhippie/ubuntu	Docker images for ubuntu	0	[OK]
docker.io	docker.io/widerplan/ubuntu	Our basic Ubuntu images.	0	[OK]

5. 删除本地镜像

如果要删除本地的镜像，可以使用 docker rmi 命令，其格式为：

```
docker rmi [选项] <镜像 1> [<镜像 2> ...]
```

注意

docker rm 命令的作用是删除容器，不要混淆。

用 ID、镜像名、摘要删除镜像，其中，<镜像>可以是镜像短 ID、镜像长 ID、镜像名或者镜像摘要。

例如有这么一些镜像：

```
# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
CentOS              latest              0584b3d2cf6d        3 weeks ago         196.5 MB
redis               alpine              501ad78535f0        3 weeks ago         21.03 MB
```


非卖品，仅供非商业用途或交流学习使用

```
docker          latest    cf693ec9b5c7    3 weeks ago     105.1 MB
nginx           latest    e43d811ce2f4    5 weeks ago     181.5 MB
```

例如这里要删除 redis:alpine 镜像，可以执行以下操作：

```
# docker rmi 501
Untagged: redis:alpine
Untagged: redis@sha256:f1ed3708f538b537eb9c2a7dd50dc90a706f7debd7e1196c9264edeea521a86d
Deleted: sha256:501ad78535f015d88872e13fa87a828425117e3d28075d0c117932b05bf189b7
Deleted: sha256:96167737e29ca8e9d74982ef2a0dda76ed7b430da55e321c071f0dbff8c2899b
Deleted: sha256:32770d1dcf835f192cafd6b9263b7b597a1778a403a109e2cc2ee866f74adf23
Deleted: sha256:127227698ad74a5846ff5153475e03439d96d4b1c7f2a449c7a826ef74a2d2fa
Deleted: sha256:1333ecc582459bac54e1437335c0816bc17634e131ea0cc48daa27d32c75eab3
Deleted: sha256:4fc455b921edf9c4aea207c51ab39b10b06540c8b4825ba57b3feed1668fa7c7
```

也可以用镜像名，也就是<仓库名>:<标签>来删除镜像。

```
# docker rmi CentOS
Untagged: CentOS:latest
Untagged: CentOS@sha256:b2f9d1c0ff5f87a4743104d099a3d561002ac500db1b9bfa02a783a46e0d366c
Deleted: sha256:0584b3d2cf6d235ee310cf14b54667d889887b838d3f3d3033acd70fc3c48b8a
Deleted: sha256:97ca462ad9eeae25941546209454496e1d66749d53dfa2ee32bf1faabd239d38
```

当然，更精确地是使用镜像摘要删除镜像。

```
# docker images --digests
REPOSITORY    TAG     DIGEST                  IMAGE ID        CREATED         SIZE
node          slim    sha256:b4f0e0bdeb578043 6e0c4c8e3913    3 weeks ago     214 MB
                      c1ea6862f0d40cc4afe32a4a5
                      82f3be235a3b164422be228
# docker rmi node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228
Untagged: node@sha256:b4f0e0bdeb578043c1ea6862f0d40cc4afe32a4a582f3be235a3b164422be228
```

像其他可以承接多个实体的命令一样，可以使用 docker images -q 来配合使用 docker rmi，这样可以成批地删除希望删除的镜像。如之前介绍过的，删除虚悬镜像的指令是：

```
# docker rmi $(docker images -q -f dangling=true)
```

前面介绍的很多过滤镜像列表的方式都可以使用。例如，需要删除所有仓库名为 redis 的镜像：

```
# docker rmi $(docker images -q redis)
```

删除所有容器：

```
# docker rm 'sudo docker ps -a -q'
```

笔 记

项目实训

【实训题目】

在 Docker Hub 上查找 Ubuntu 镜像，下载并运行，查看本地镜像列表，练习完成后删除 Ubuntu 镜像。

【实训目的】

1. 掌握 Docker 镜像的线上运维，如搜索、下载镜像。
2. 掌握 Docker 镜像的线下运维，如运行、列出镜像、删除。

【实训内容】

1. 搜索 Docker Hub 上的 Ubuntu 镜像。
2. 下载搜索到的镜像。
3. 镜像下载到本地后，查看本地的镜像列表。
4. 运行下载好的镜像。
5. 练习完成，删除镜像。

任务 2.2　Docker 镜像定制

任务描述

使用 Docker Dockerfile 进行镜像定制，目前定制的方法有以下两种。

① 手动修改容器内容，生成 Docker 镜像。

② Dockerfile 包含创建镜像所需要的全部指令。基于在 Dockerfile 中的指令，这里可以使用 docker build 命令来创建镜像。通过减少镜像和容器的创建过程来简化部署。

知识学习

1. Dockerfile 简介

镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，是以镜像为基础层，在其基础上加一层作为容器运行时的存储层。

创建镜像有两种方法，一是用 commit 命令，二是用 Dockerfile 方法（这个更常用，在下面介绍）。

在介绍 commit 命令前，首先回顾一下对代码的版本控制。当修改代码后，可以使用 commit 命令变更到版本服务器上。对于容器，当创建容器后，如果后面对容器做了修改，就可以利用 commit 命令将修改提交为一个新的镜像。

Dockerfile 是由一系列命令和参数构成的脚本，这些命令应用于基础镜像并最终创建一个新的镜像。它们简化了从头到尾的流程，并极大地简化了部署工作。Dockerfile 从 FROM 命令开始，紧接着跟随着各种方法、命令和参数。其产出为一个新的可以用于创建容器的镜像。

2. Dockerfile 指令详解

(1) COPY

格式：

```
COPY <源路径>... <目标路径>
COPY ["<源路径>",... "<目标路径>"]
```

和 RUN 指令一样，也有两种格式，一种类似于命令行，一种类似于函数调用。

COPY 指令将从构建上下文目录中<源路径>的文件/目录复制到新的一层镜像内的<目标路径>位置，如：

```
COPY package.json /usr/src/app/
```

<源路径>可以是多个，甚至可以是通配符，其通配符规则要满足 Go 的filepath.Match 规则，如：

```
COPY hom* /mydir/COPY hom?.txt /mydir/
```

<目标路径>可以是容器内的绝对路径，也可以是相对于工作目录的相对路径（工作目录可以用 WORKDIR 指令来指定）。目标路径不需要事先创建，如果目录不存在，那么会在复制文件前先行创建缺失目录。

此外，还需要注意一点，使用 COPY 指令，源文件的各种元数据都会保留。比如读、写、执行权限，文件变更时间等。这个特性对于镜像定制很有用。特别是构建相关文件都在使用 Git 进行管理的时候。

(2) ADD

ADD 指令和 COPY 的格式及性质基本一致，只是在 COPY 的基础上增加了一些功能。

比如<源路径>，可以是一个 URL，这种情况下，Docker 引擎会试图去下载这个链接的文件并放到<目标路径>去。下载后的文件权限自动设置为 600，如果这并不是想要的权限，那么还需要增加额外的一层 RUN 进行权限调整。另外，如果下载的是个压缩包，那么需要解压缩，此时也一样需要额外的一层 RUN 指令进行解压缩。所以不如直接使用 RUN 指令，然后使用 wget 或者 curl 工具下载，处理权限，解压缩，然后清理无用文件更合理。因此，这个功能其实并不实用，而且不推荐使用。

如果<源路径>为一个 tar 压缩文件，且压缩格式为 gzip、bzip2 及 xz 的情况下，那么 ADD 指令将会自动解压缩这个压缩文件到<目标路径>。

在某些情况下，这个自动解压缩的功能非常有用，如官方镜像 Ubuntu 中：

```
FROM scratch ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /...
```

但在某些情况下，如果希望复制某个压缩文件进去，而不解压缩，那么就不可以使用 ADD 命令了。

Docker 官方的最佳实践文档要求尽可能地使用 COPY，因为 COPY 的语义很明确，就是复制文件而已，而 ADD 则包含了更复杂的功能，其行为也不一定很清晰。最适合使用 ADD 的场合，就是所提及的需要自动解压缩的场合。

注意

ADD 指令会令镜像构建缓存失效，从而可能会令镜像构建变得比较缓慢。

因此在 COPY 和 ADD 指令中选择的时候，可以遵循这样的原则，所有的文件复制均使用 COPY 指令，仅在需要自动解压缩的场合使用 ADD 指令。

（3）CMD

CMD 指令的格式和 RUN 相似，也是两种格式：

shell 格式：

```
CMD <命令>
```

exec 格式：

```
CMD ["可执行文件", "参数 1", "参数 2"...]
```

参数列表格式：CMD ["参数 1", "参数 2"...]。在指定了 ENTRYPOINT 指令后，用 CMD 指定具体的参数。

之前介绍容器的时候曾经说过，Docker 不是虚拟机，容器就是进程。既然是进程，那么在启动容器的时候需要指定所运行的程序及参数。CMD 指令就是用于指定默认的容器主进程的启动命令的。

在运行时可以指定新的命令来替代镜像设置中的这个默认命令，比如，Ubuntu 镜像默认的 CMD 是/bin/bash，如果直接 docker run -it ubuntu，则会直接进入 bash。也可以在运行时指定运行别的命令，如 docker run -it ubuntu cat /etc/os-release。这就是用 cat/etc/os-release 命令替换了默认的/bin/bash 命令，输出了系统版本信息。

在指令格式上，一般推荐使用 exec 格式，这类格式在解析时会被解析为 json 数组，因此一定要使用双引号，而不要使用单引号。

如果使用 shell 格式，那么实际的命令会被包装为“sh　c”的参数的形式进行执行。比如：

```
CMD echo $HOME
```

在实际执行中，会将其变更为：

```
CMD [ "sh", "-c", "echo $HOME" ]
```

这就是为什么可以使用环境变量的原因，因为这些环境变量会被 shell 进行解析处理。

提到 CMD，就不得不提容器中的应用在前台执行和后台执行的问题。这是初学者常出现的一个混淆。

Docker 不是虚拟机，容器中的应用都应该在前台执行，而不是像虚拟机、物理机那样，用 upstart/systemd 去启动后台服务，容器内没有后台服务的概念。

一些初学者将 CMD 写为：

```
CMD service nginx start
```

然后发现容器执行后就立即退出了，甚至在容器内使用 systemctl 命令，结果却发现根本执行不了。这就是因为没有搞明白前台、后台的概念，没有区分容器和虚拟机的差异，依旧在以传统虚拟机的角度去理解容器。

对于容器而言，其启动程序就是容器应用进程，容器就是为了主进程而存在的。主进程退出，容器就失去了存在的意义，从而退出，其他辅助进程不是它需要关心的东西。

而使用 service nginx start 命令，则是希望用 upstart 来以后台守护进程的形式启动 nginx

服务。刚才说了，CMD service nginx start 会被理解为 CMD ["sh", "-c", "service nginx start"]，因此主进程实际上是 sh。那么当 service nginx start 命令结束后，sh 也就结束了。sh 作为主进程退出了，自然就会令容器退出。

正确的做法是直接执行 nginx 可执行文件，并且要求以前台形式运行。比如：

```
CMD ["nginx" "-g" "daemon off;"]
```

(4) ENTRYPOINT

ENTRYPOINT 的格式和 RUN 指令格式一样，分为 exec 格式和 shell 格式。

ENTRYPOINT 的目的和 CMD 一样，都是在指定容器启动程序及参数。

ENTRYPOINT 在运行时也可以替代，不过比 CMD 要略显烦琐，需要通过 docker run 的参数“--entrypoint”来指定。

当指定了 ENTRYPOINT 后，CMD 的含义就发生了改变，不再是直接运行其命令，而是将 CMD 的内容作为参数传给 ENTRYPOINT 指令。换句话说，实际执行时将变为：

```
<ENTRYPOINT> "<CMD>"
```

那么有了 CMD 后，为什么还要有 ENTRYPOINT 呢？这种<ENTRYPOINT> "<CMD>"有什么好处呢？来看一个场景。

>> 场景： 让镜像变成像命令一样使用。

假设需要一个知道自己当前公网 IP 的镜像，那么可以先用 CMD 来实现：

```
FROM ubuntu:16.04
RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*
CMD [ "curl", "-s", "http://ip.cn" ]
```

假如使用 docker build -t myip 来构建镜像，如果需要查询当前公网 IP，只需要执行以下命令：

```
# docker run myip
```

当前 IP：61.148.226.66 来自：北京市 联通

这么看起来好像可以直接把镜像当作命令使用了，不过命令总有参数，如果希望加参数呢？例如，从上面的 CMD 中可以看到实质的命令是 curl，如果希望显示 HTTP 头信息，就需要加上“-i”参数，那么可以直接加“-i”参数给 docker run myip 吗？

```
$ docker run myip -i
docker: Error response from daemon: invalid header field value "oci runtime error: container_linux.go:247:
starting container process caused \"exec: \\\"-i\\\": executable file not found in $PATH\"\n".
```

此时可以看到可执行文件找不到的报错，即 executable file not found。之前说过，跟在镜像名后面的是 command，运行时会替换 CMD 的默认值。因此这里的“-i”替换了原来的 CMD，而不是添加在原来的 curl -s http://ip.cn 后面。而“-i”根本不是命令，所以自然找不到。

那么如果希望加入“-i”参数，就必须重新完整地输入这个命令：

```
$ docker run myip curl -s http://ip.cn -i
```

这显然不是很好的解决方案，而使用 ENTRYPOINT 就可以解决这个问题。现在重新用 ENTRYPOINT 来实现这个镜像：

```
FROM ubuntu:16.04
RUN apt-get update \
    && apt-get install -y curl \
    && rm -rf /var/lib/apt/lists/*
ENTRYPOINT [ "curl", "-s", "http://ip.cn" ]
```

这次再来尝试直接使用 docker run myip -i：

```
$ docker run myip
```

当前 IP：61.148.226.66 来自：北京市 联通

可以看到，这次成功了。这是因为当存在 ENTRYPOINT 后，CMD 的内容将会作为参数传给 ENTRYPOINT，而这里的“-i”就是新的 CMD，因此会作为参数传给 curl，从而达到了预期的效果。

（5）ENV

格式有两种：

```
ENV <key><value>
ENV <key1>=<value1><key2>=<value2>...
```

这个指令很简单，就是设置环境变量而已，无论是后面的其他指令，如 RUN，还是运行时的应用，都可以直接使用这里定义的环境变量。

```
ENV VERSION=1.0 DEBUG=on \
    NAME="Happy Feet"
```

这个例子中演示了如何换行，以及对含有空格的值用双引号括起来的方法，这和 shell 下的行为是一致的。

定义了环境变量，那么在后续的指令中就可以使用这个环境变量。比如在官方 node 镜像 Dockerfile 中，就有类似这样的代码：

```
ENV NODE_VERSION 7.2.0
RUN curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/node-v$NODE_VERSION-linux-x64.tar.xz" \
  && curl -SLO "https://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \
  && gpg --batch --decrypt --output SHASUMS256.txt SHASUMS256.txt.asc \
  && grep " node-v$NODE_VERSION-linux-x64.tar.xz\$" SHASUMS256.txt | sha256sum -c - \
  && tar -xJf "node-v$NODE_VERSION-linux-x64.tar.xz" -C /usr/local --strip-components=1 \
  && rm "node-v$NODE_VERSION-linux-x64.tar.xz" SHASUMS256.txt.asc SHASUMS256.txt \
  && ln -s /usr/local/bin/node /usr/local/bin/nodejs
```

在这里先定义了环境变量 NODE_VERSION，其后的 RUN 这层里，多次使用 $NODE_VERSION 来进行操作定制。可以看到，将来升级镜像构建版本的时候，只需要更新版本 7.2.0 即可，Dockerfile 构建维护变得更轻松了。

下列指令可以支持环境变量展开：ADD、COPY、ENV、EXPOSE、LABEL、USER、

WORKDIR、VOLUME、STOPSIGNAL、ONBUILD。

用户可以从这个指令列表里感觉到，环境变量可以使用的地方很多，很强大。环境变量，可以让一份 Dockerfile 制作更多的镜像，只需使用不同的环境变量即可。

（6）ARG

格式：

```
ARG <参数名>[=<默认值>]
```

ARG 和 ENV 一样，都是设置环境变量。所不同的是，ARG 所设置的是构建环境的环境变量，在将来容器运行时是不会存在这些环境变量的。但是不要因此就使用 ARG 保存密码之类的信息，因为 docker history 还是可以看到所有值的。

Dockerfile 中的 ARG 指令是定义参数名称及定义其默认值的。该默认值可以在构建命令 docker build 中用--build-arg <参数名>=<值>来覆盖。

对于 1.13 之前的版本，要求--build-arg 中的参数名必须在 Dockerfile 中用 ARG 定义过了，换句话说，就是--build-arg 指定的参数必须在 Dockerfile 中使用了。如果对应参数没有被使用，则会报错并退出构建。从 1.13 版本开始，这种严格的限制被放开，不再报错退出，而是显示警告信息，并继续构建。这对于在 CI 系统中用同样的构建流程构建不同的 Dockerfile 时比较有帮助，避免构建命令必须根据每个 Dockerfile 的内容修改。

（7）VOLUME

格式：

```
VOLUME ["<路径 1>", "<路径 2>"...]
VOLUME <路径>
```

容器运行时应该尽量保持容器存储层不发生写操作，数据库类需要保存动态数据的应用，其数据库文件应该保存于卷（Volume）中，后面的章节会进一步介绍 Docker 卷的概念。为了防止运行时用户忘记将动态文件所保存的目录挂载为卷，在 Dockerfile 中，可以事先指定某些目录挂载为匿名卷，这样在运行时如果用户不指定挂载，其应用也可以正常运行，不会向容器存储层写入大量数据。

```
VOLUME /data
```

这里的“/data”目录就会在运行时自动挂载为匿名卷，任何向“/data”中写入的信息都不会记录进容器存储层，从而保证了容器存储层的无状态化。当然，运行时可以覆盖这个挂载设置。比如：

```
docker run -d -v mydata:/data xxxx
```

在这行命令中就将 mydata 这个卷挂载到了“/data”这个位置，替代了 Dockerfile 中定义的匿名卷的挂载配置。

（8）EXPOSE

格式：

```
EXPOSE <端口 1> [<端口 2>...]
```

EXPOSE 指令声明运行时容器提供的服务端口，这只是一个声明，在运行时并不会因为这个声明应用就会开启这个端口的服务。在 Dockerfile 中写入这样的声明有两个好处：

版权所有，严禁传播，违者自负法律责任！

一个是帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射；另一个好处则是在运行时使用随机端口映射时，也就是 docker run -P 时，会自动随机映射 EXPOSE 的端口。

此外，在早期的 Docker 版本中还有一个特殊的用处。以前，所有容器都运行于默认连接网络中，因此容器互相之间都可以直接访问，这样存在一定的安全性问题。于是有了一个 Docker 引擎参数--icc=false，当指定该参数后，容器间将默认无法互访，除非相互使用了--links 参数的容器才可以互通，并且只有镜像中 EXPOSE 所声明的端口才可以被访问。这个--icc=false 的用法，在引入了 docker network 后已经基本不用了，通过自定义网络可以很轻松地实现容器间的互联与隔离。

用户要将 EXPOSE 和在运行时使用的-p <宿主端口>:<容器端口>区分开来。“-p”可映射宿主端口和容器端口，换句话说，就是将容器的对应端口服务公开给外界访问，而 EXPOSE 仅仅是声明容器打算使用什么端口而已，并不会自动在宿主进行端口映射。

（9）WORKDIR

格式：

```
WORKDIR <工作目录路径>
```

使用 WORKDIR 指令可以来指定工作目录（或者称为当前目录），以后各层的当前目录就被改为指定的目录。该目录需要已经存在，WORKDIR 并不会帮开发者建立目录。

初学者常犯的一些错误其中之一就是把 Dockerfile 等同于 shell 脚本来书写，这种错误还可能会导致出现下面这样的错误：

```
RUN cd /appRUN echo "hello" > world.txt
```

运行后就会发现找不到/app/world.txt 文件，或者其内容不是 hello。原因其实很简单，在 shell 中，连续两行是同一个进程执行环境，因此前一个命令修改的内存状态会直接影响后一个命令；而在 Dockerfile 中，这两行 RUN 命令的执行环境根本不同，是两个完全不同的容器。这就是对 Dokerfile 构建分层存储的概念不了解所导致的错误。

之前说过，每一个 RUN 都启动一个容器，执行命令，然后提交存储层文件变更。第一层 RUN cd /app 的执行仅仅是当前进程的工作目录变更，一个内存上的变化而已，其结果不会造成任何文件变更。而到第二层的时候，启动的是一个全新的容器，跟第一层的容器完全没关系，自然不可能继承前一层构建过程中的内存变化。

因此，如果需要改变以后各层的工作目录的位置，那么应该使用 WORKDIR 指令。

（10）USER

格式：

```
USER <用户名>
```

USER 指令和 WORKDIR 相似，都可以改变环境状态并影响以后的层。WORKDIR 是改变工作目录，USER 则是改变之后的层执行 RUN、CMD 及 ENTRYPOINT 这类命令的身份。

当然，和 WORKDIR 一样，USER 只是帮助开发者切换到指定用户而已，这个用户必须是事先建立好的，否则无法切换。

```
RUN groupadd -r redis && useradd -r -g redis redisUSER redisRUN [ "redis-server" ]
```

如果以 root 执行的脚本在执行期间希望改变身份，比如希望以某个已经建立好的用

版权所有，严禁传播，违者自负法律责任！

户来运行某个服务进程，则不要使用 su 或者 sudo，这些都需要比较麻烦的配置，而且在 TTY 缺失的环境下经常出错。此时建议使用 gosu，用户可以从其项目网站看到进一步的信息，网址为https://github.com/tianon/gosu。

```
#建立 redis 用户，并使用 gosu 换另一个用户执行命令
RUN groupadd -r redis && useradd -r -g redis redis
#下载 gosuRUN
wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/1.7/gosu-amd64" \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true
 #设置 CMD，并以另外的用户执行
CMD [ "exec", "gosu", "redis", "redis-server" ]
```

任务实施

1. 利用 commit 理解镜像构成

镜像是容器的基础，每次执行 docker run 的时候都会指定哪个镜像作为容器运行的基础。在之前的例子中，所使用的都是来自于 Docker Hub 的镜像。直接使用这些镜像可以满足一定的需求，而当这些镜像无法直接满足需求时，就需要定制这些镜像。接下来就将讲解如何定制镜像。

镜像是多层存储，每一层是在前一层的基础上进行的修改；容器同样也是多层存储，是在以镜像为基础层，并在其基础上加一层作为容器运行时的存储层。

现在以定制一个 Web 服务器为例子来讲解镜像是如何构建的。

```
# docker run --name webserver -d -p 80:80 nginx
```

这条命令会用 nginx 镜像启动一个容器，命名为 webserver，并且映射了 80 端口，这样可以用浏览器去访问这个 Nginx 服务器。

直接用浏览器访问的话，会看到默认的 Nginx 欢迎页面，如图 2-2-1 所示。

图 2-2-1
Nginx 欢迎页面

现在，假设开发者非常不喜欢这个欢迎页面，希望改成欢迎 Docker 的文字，则可以使用 docker exec 命令进入容器，修改其内容。

```
# docker exec -it webserver bash
root@3729b97e8226:/# echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
root@3729b97e8226:/# exit
```

此时以交互式终端方式进入 webserver 容器，并执行了 bash 命令，也就是获得一个可操作的 shell，然后用<h1>Hello, Docker!</h1>覆盖/usr/share/nginx/html/index.html 的内容。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

现在再刷新浏览器的话，会发现内容改变了，如图 2-2-2 所示。

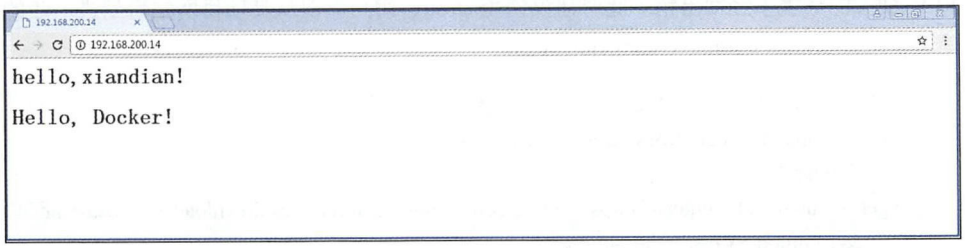


图 2-2-2
修改后的页面

修改了容器的文件，也就是改动了容器的存储层。用户可以通过 docker diff 命令看到具体的改动。

```
# docker diff webserver
C /root
A /root/.bash_history
C /run
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
C /var
C /var/cache
C /var/cache/nginx
A /var/cache/nginx/client_temp
A /var/cache/nginx/fastcgi_temp
A /var/cache/nginx/proxy_temp
A /var/cache/nginx/scgi_temp
A /var/cache/nginx/uwsgi_temp
```

现在定制好了变化，希望能将其保存下来以形成镜像。

要知道，当运行一个容器的时候（如果不使用卷的话），做的任何文件修改都会被记录于容器存储层里。而 Docker 提供了一个 docker commit 命令，可以将容器的存储层保存下来成为镜像。在原有镜像的基础上，再叠加上容器的存储层，构成新的镜像，以后运行这个新镜像的时候，就会拥有原有容器最后的文件变化。语法格式为：

```
docker commit [选项] <容器 ID 或容器名> [<仓库名>[:<标签>]]
```

可以用下面的命令将容器保存为镜像：

```
# docker commit \
    --author "Tao Wang <twang2218@gmail.com>" \
    --message "修改了默认网页" \
    webserver \
    nginx:v2
sha256:07e33465974800ce65751acc279adc6ed2dc5ed4e0838f8b86f0c87aa1795214
```

用户可以在 docker images 中看到这个新定制的镜像：

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
# docker images nginx
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
nginx               v2                  07e334659748        9 seconds ago       181.5 MB
nginx               1.11                05a60462f8ba        12 days ago         181.5 MB
nginx               latest              e43d811ce2f4        4 weeks ago         181.5 MB
```

用户还可以用 docker history 具体查看镜像内的历史记录，如果比较 nginx:latest 的历史记录，就会发现新增了刚刚提交的这一层。

```
# docker history nginx
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
07e334659748        54 seconds ago      nginx -g daemon off;                            95 B
e43d811ce2f4        4 weeks ago         /bin/sh -c #（nop）                              0 B
                                        CMD ["nginx" "-g" "daemon
<missing>           4 weeks ago         /bin/sh -c #（nop）EXPOSE                        0 B
                                        443/tcp 80/tcp
<missing>           4 weeks ago         /bin/sh -c ln -sf /dev/stdout                   22 B
                                        /var/log/nginx/
<missing>           4 weeks ago         /bin/sh -c apt-key adv                          58.46 MB
                                        --keyserver hkp://pgp.
<missing>           4 weeks ago         /bin/sh -c #（nop） ENV                          0 B
                                        NGINX_VERSION=1.11.5-1
<missing>           4 weeks ago         /bin/sh -c #（nop）                              0 B
                                        MAINTAINER NGINX Docker Ma
<missing>           4 weeks ago         /bin/sh -c #（nop）CMD                           0 B
                                        ["/bin/bash"]
<missing>           4 weeks ago         /bin/sh -c #（nop）ADD file:                     123 MB
                                        23aa4f893e3288698c
```

新的镜像定制好后，可以来运行这个镜像。

```
# docker run --name web2 -d -p 81:80 nginx
```

这里命名新的服务为 web2，并且映射到 81 端口。如果是 Docker for Mac/Windows 或 Linux 桌面，就可以直接访问http://localhost:81看到结果，其内容应该和之前修改后的 webserver 一样。

至此，第一次完成了定制镜像，使用的是 docker commit 命令，通过手动操作给旧的镜像添加了新的一层，形成新的镜像。至此，用户对镜像多层存储应该有了更直观的感觉。

注意

慎用 docker commit。使用 docker commit 命令虽然可以比较直观地帮助理解镜像分层存储的概念，但是实际环境中并不会这样使用。

首先，如果仔细观察之前的 docker diff webserver 的结果，开发者就会发现除了真正想要修改的/usr/share/nginx/html/index.html 文件外，由于命令的执行，还有很多文件被改动或添加了。这还仅仅是最简单的操作，如果是安装软件包、编译构建，那么会有大量的无关内容被添加进来，如果不小心清理，将会导致镜像极为臃肿。

此外，使用 docker commit 意味着所有对镜像的操作都是黑箱操作，生成的镜像也被

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

称为黑箱镜像，除了制作镜像的人知道执行过什么命令、怎么生成的镜像外，别人根本无从得知。而且，即使是这个制作镜像的人，时间长后也无法记清具体操作的步骤。虽然使用 docker diff 或许可以得到一些线索，但是远远达不到可以确保生成一致镜像的地步。这种黑箱镜像的维护工作是非常痛苦的。

而且，回顾之前提及的镜像所使用的分层存储的概念，除当前层外，之前的每一层都是不会发生改变的，换句话说，任何修改的结果仅仅是在当前层进行标记、添加、修改，而不会改动上一层。如果使用 docker commit 制作镜像，以及进行后期修改，那么每一次修改都会让镜像更加臃肿，所删除的上一层的东西并不会丢失，会一直如影随形地跟着这个镜像，即使根本无法访问到它。

docker commit 命令还有一些特殊的应用场合，比如被入侵后保存现场等。但是，不要使用 docker commit 定制镜像，定制行为应该使用 Dockerfile 来完成。下面就来讲述如何使用 Dockerfile 定制镜像。

2. 使用 Dockerfile 定制镜像

① 创建镜像目录。

```
# mkdir static_web
# cd static_web/
# touch Dockerfile
# ll
total 0
-rw-r--r--. 1 root root 0 Dec 25 13:59 Dockerfile
```

② 编辑新镜像内容。

```
# vi Dockerfile
# Version: 0.0.1
FROM ubuntu/14.04.3.tar
MAINTAINER James Turnbull "james@example.com"
RUN apt-get update
RUN apt-get install -y nginx
RUN echo 'Hi, I am in your container' \
        >/var/www/html/index.html
EXPOSE 80
```

③ 执行命令来构建新镜像。

```
# sudo docker build -t="jamtur01/static_web" .
-t：构建一个名称
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu/14.04.3.tar
 ---> 8e1acf623cb6
Step 2 : MAINTAINER James Turnbull "james@example.com"
 ---> Running in 99c1fff85acf
 ---> bc46e1a77642
Removing intermediate container 99c1fff85acf
Step 3 : RUN apt-get update && apt-get install -y nginx
 ---> Running in 7a3788bdfef1
Err http://archive.ubuntu.com trusty InRelease
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

④ 如果有错，可以进入最后成功的步骤创建新容器。

```
# docker run -t -i bc46e1a77642 /bin/bash
root@c463ce8417c1:/# vi   /etc/hosts
```

⑤ 执行命令，创建镜像的正确结果。

```
# sudo docker build -t="jamtur01/static_web" .
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM ubuntu/14.04.3.tar
 ---> c32bb61e14d4
Step 2 : MAINTAINER James Turnbull "james@example.com"
 ---> Running in 97f96f4aa2ee
 ---> a4c5787baff4
Removing intermediate container 97f96f4aa2ee
Step 3 : RUN apt-get update
 ---> Running in 332213654e40
Ign http://archive.ubuntu.com trusty InRelease
Get:1 http://archive.ubuntu.com trusty-updates InRelease [65.9 kB]
Get:2 http://archive.ubuntu.com trusty-security InRelease [65.9 kB]
Hit http://archive.ubuntu.com trusty Release.gpg
Hit http://archive.ubuntu.com trusty Release
Get:3 http://archive.ubuntu.com trusty-updates/main Sources [480 kB]
Processing triggers for libc-bin （2.19-0ubuntu6.6）...
 ---> 4a7703adb76c
Removing intermediate container 40537c65edbc
Step 5 : RUN echo 'Hi, I am in your container'        >/var/www/html/index.html
 ---> Running in abab0b5cc08a
 ---> e4764ec07a72
Removing intermediate container abab0b5cc08a
Step 6 : EXPOSE 80
 ---> Running in 95ac9d28b646
 ---> 0538423a4072
Removing intermediate container 95ac9d28b646
Successfully built 0538423a4072
```

⑥ Dockerfile 和构建缓存。

可以使用--no-cache 忽略 Dockerfile 的构建缓存：

```
# sudo docker build --no-cache -t="jamtur01/static_web" .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu/14.04.3.tar
 ---> 8e1acf623cb6
Step 2 : MAINTAINER James Turnbull "james@example.com"
 ---> Running in 9dd8042f0ace
 ---> b12b41d7d515
Removing intermediate container 9dd8042f0ace
Step 3 : RUN apt-get update && apt-get install -y nginx
 ---> Running in 9ed141f6d333
 ---> 3d612a28f61b
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
.....
Removing intermediate container 300c41abf2a4
Step 5 : RUN echo 'Hi, I am in your container'      >/var/www/html/index.html
 ---> Running in 2375fd4c6625
 ---> 9a85a386547b
Removing intermediate container 2375fd4c6625
Step 6 : EXPOSE 80
 ---> Running in e41d5977b4c0
 ---> aa2a541b59ce
Removing intermediate container e41d5977b4c0
Successfully built aa2a541b59ce
```

项目实训

【实训题目】

定制一个 Web 网站，在网站页面上打印“Hello world!”。

【实训目的】

1. 掌握 commit 命令的使用方法。
2. 掌握 Dockerfile 的编写。

【实训内容】

使用 commit 定制镜像：

1. 下载 Nginx 镜像并运行容器。
2. 进入 Nginx 容器，修改 Nginx 主页。
3. 使用 commit 命令保存修改过的镜像。
4. 使用 Dockerfile 定制镜像。
5. 创建镜像目录。
6. 编写 Dockerfile 文件。
7. 使用 Dockerfile 构建新镜像。

笔 记

任务 2.3　Docker 容器管理

任务描述

1. 容器的启动、运行、停止、删除等基本外部操作。
2. 容器内部的查询、修改，安装软件等基本操作。
3. 容器的日志运维排错。

知识学习

1. 容器介绍

容器是 Docker 的又一核心概念。简单地说，容器是独立运行的一个或一组应用，以

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

及它们的运行态环境，可以简单从下面 4 个方面来阐述。

① Docker 利用容器来运行应用。

② 容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。

③ 可以把容器看作是一个简易版的 Linux 环境（包括 root 用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

④ 镜像是只读的，容器在启动的时候创建一层可写层作为最上层。

Docker 提供了轻量级的虚拟化，它几乎没有额外开销，这个特性非常酷。

首先，用户在享有 Docker 带来的虚拟化层的时候不用担心因此而带来的额外开销。然后，开发者能在单一机器上启动更多数量的容器。最后，它的一个强有力的影响是容器的启与停都能在几秒钟完成。

2. Dockerfile、Docker 镜像和 Docker 容器的关系

Dockerfile 是软件的原材料，Docker 镜像是软件的交付品，而 Docker 容器则可以认为是软件的运行态。从应用软件的角度来看，Dockerfile、Docker 镜像与 Docker 容器分别代表软件的 3 个不同阶段：Dockerfile 面向开发，Docker 镜像成为交付标准，Docker 容器则涉及部署与运维。三者缺一不可，合力充当 Docker 体系的基石。

简单来讲，Dockerfile 构建出 Docker 镜像，通过 Docker 镜像运行 Docker 容器。

这里可以从 Docker 容器的角度来反推三者的关系。图 2-3-1 所示为 Docker 容器三方的关系。

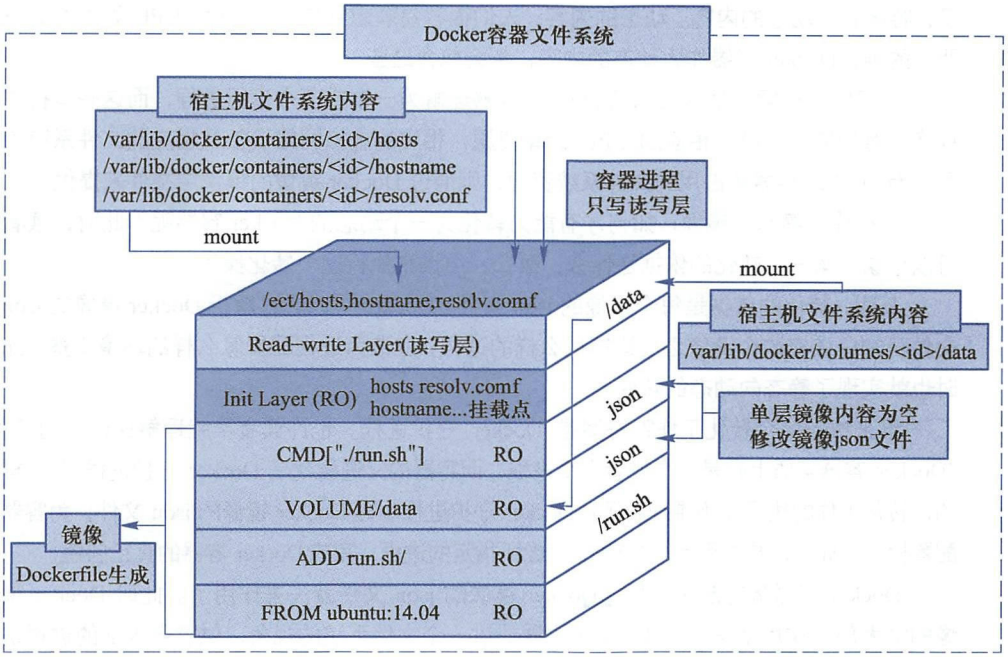


图 2-3-1
Docker 容器的
三方关系

假设这个容器的镜像通过以下 Dockerfile 构建而得：

```
FROM ubuntu:14.04
ADD run.sh /
VOLUME /data
CMD ["./run.sh"]
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

笔 记

（1）Dockerfile 与 Docker 镜像

首先，结合图 2-3-1 来看 Dockerfile 与 Docker 镜像之间的关系。

- FROM ubuntu:14.04：设置基础镜像，此时会使用基础镜像 ubuntu:14.04 的所有镜像层，为简单起见，图 2-3-1 中将其作为一个整体展示。
- ADD run.sh /：将 Dockerfile 所在目录的文件 run.sh 加至镜像的根目录，此时，新一层的镜像只有一项内容，即根目录下的 run.sh。
- VOLUME /data：设定镜像的 VOLUME，此 VOLUME 在容器内部的路径为 /data。需要注意的是，此时并未在新一层的镜像中添加任何文件，即构建出的此层镜像中文件为空，但更新了镜像的 json 文件，以便通过此镜像启动容器时获取这方面的信息。
- CMD ["./run.sh"]：设置镜像的默认执行入口，此命令同样不会在新建镜像中添加任何文件，仅仅在上一层镜像 json 文件的基础上更新新建镜像的 json 文件。

因此，通过以上分析可知，以上的 Dockerfile 可以构建出一个新的镜像，包含 4 个镜像层，每一条命令会和一个镜像层对应，镜像之间会存在父子关系。图中很清楚地表明了这些关系。

（2）Docker 镜像与 Docker 容器的关系

Docker 镜像是 Docker 容器运行的基础，没有 Docker 镜像，就不可能有 Docker 容器，这也是 Docker 的设计原则之一。

可以理解的是：Docker 镜像毕竟是镜像，属于静态的内容。而 Docker 容器就不一样了，容器属于动态的内容。动态的内容，人们很容易联想到进程、内存、CPU 等之类的东西。的确，Docker 容器作为动态的内容，都会包含这些。

为了便于理解，读者可以把 Docker 容器理解为一个或多个运行进程，而这些运行进程将占有相应的内存、相应的 CPU 计算资源、相应的虚拟网络设备及相应的文件系统资源。而 Docker 容器所占用的文件系统资源，则通过 Docker 镜像的镜像层文件来提供。

那么作为静态的镜像，如何才有能力转化为一个动态的 Docker 容器呢？此时，读者可以想象：第一，转化的依据是什么；第二，由谁来执行这个转化操作。

其实，转化的依据是每个镜像的 json 文件，Docker 可以通过解析 Docker 镜像的 json 文件获知应该在这个镜像之上运行什么样的进程，应该为进程配置怎么样的环境变量，此时也就实现了静态向动态的转变。

谁来执行这个转化工作？答案是 Docker 守护进程。也许读者早就理解这样一句话："Docker 容器实质上就是一个或者多个进程，而容器的父进程就是 Docker 守护进程。"这样的，转化工作的执行就不难理解了：Docker 守护进程手握 Docker 镜像的 json 文件，为容器配置相应的环境，并真正运行 Docker 镜像所指定的进程，完成 Docker 容器的真正创建。

Docker 容器运行起来之后，Docker 镜像的 json 文件就失去作用了。此时 Docker 镜像的绝大部分作用就是：为 Docker 容器提供一个文件系统的视角，供容器内部的进程访问文件资源。

再次回到图 2-3-1，再来看看容器和镜像之间的一些特殊关系。首先，之前已经提及 Docker 镜像是分层管理的，管理 Docker 容器的时候，Docker 镜像仍然是分层管理的。由于此时动态的容器中已经存在进程，进程就会对文件系统视角内的文件进行读写操作，因此就会涉及一个问题：容器是否会篡改 Docker 镜像的内容。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

答案自然是不会的。统一来讲，正如图 2-3-1 所示，所有的 Docker 镜像层对于容器来说都是只读的，容器对于文件的写操作绝对不会作用在镜像中。

既然如此，实现的原理就很重要，究其根本：Docker 守护进程会在 Docker 镜像的最上层之上再添加一个可读写层，容器所有的写操作都会作用到这一层中。而如果 Docker 容器需要写底层 Docker 镜像中的文件，那么此时就会涉及一个名称为 Copy-on-Write 的机制，即 aufs 等联合文件系统保证：首先将此文件从 Docker 镜像层中复制至最上层的可读写层，然后容器进程再对读写层中的副本进行写操作。对于容器进程来讲，它只能看到最上层的文件。

那最后再来说说：在 Docker 容器的文件系统视角中，到底是不是存在一些内容不是存储于 Docker 镜像中的？这次的答案依旧是肯定的。

再次重申一点，Docker 镜像中存储的都是一些静态文件，这些文件原则上应该和容器具体信息及主机信息完全耦合。那么，在 Docker 容器中不存储于 Docker 镜像中的内容主要有以下几种。

① /proc 及/sys 等虚拟文件系统的内容。

② 容器的 hosts 文件、hostname 文件及 resolv.conf 文件，这些是具体环境的信息，原则上的确不应该被归入镜像。

③ 容器的 Volume 路径，这部分的视角来源于从宿主机上挂载到容器内部的路径。

④ 部分的设备文件。

任务实施

1. 容器外部操作

（1）启动容器

启动容器有两种方式，一种是基于镜像新建一个容器并启动，另外一种是将在终止状态（stopped）的容器重新启动。

新建并启动所需要的命令主要为 docker run。

例如，下面的命令输出一个“Hello world”，之后终止容器。

```
# sudo docker run ubuntu:14.04 /bin/echo 'Hello world'
Hello world
```

这与在本地直接执行/bin/echo 'hello world' 几乎没有任何区别。

下面的命令则启动一个 bash 终端，允许用户进行交互。

```
# sudo docker run -t -i ubuntu:14.04 /bin/bash
root@af8bae53bdd3:/#
```

其中，“-t”选项让 Docker 分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上，“-i”则让容器的标准输入保持打开。

在交互模式下，用户可以通过所创建的终端来输入命令，例如：

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

当利用 docker run 来创建容器时，Docker 在后台运行的标准操作如下。

① 检查本地是否存在指定的镜像，不存在就从公有仓库下载。

② 利用镜像创建并启动一个容器。

③ 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层。

④ 从宿主机配置的网桥接口中桥接一个虚拟接口到容器中去。

⑤ 从地址池配置一个 IP 地址给容器。

⑥ 执行用户指定的应用程序。

⑦ 执行完毕后容器被终止。

⑧ 启动已终止容器。

可以利用 docker start 命令直接将一个已经终止的容器启动运行。

```
# docker ps -a
CONTAINER ID  IMAGE          COMMAND                  CREATED         STATUS                      PORTS                    NAMES
6f3bbb08549f  ubuntu:14.04   "/bin/echo 'Hello wor"   About a minute ago  Exited (0) About a minute ago                    focused_northcutt
896ba7a2099d  nginx          "nginx -g 'daemon off"   2 hours ago     Up 2 hours                  443/tcp, 0.0.0.0:81->80/tcp  web2
5a5fb3a18ea8  nginx          "nginx -g 'daemon off"   4 hours ago     Up 4 hours                  0.0.0.0:80->80/tcp, 443/tcp  webserver
3bdab40bc301  ubuntu:latest  "ip add show"            11 hours ago    Created                                              pedantic_wescoff
ea59a492cf02  192.168.200.11:5000/ubuntu/14.04.3.tar  "latest ip add show"  11 hours ago  Created                                  furious_dubinsky
cd507e407c1c  jamtur01/consul  "/usr/sbin/consul age"  11 hours ago  Exited (1) 10 hours ago                              tiny_goldstine
# docker start cd507e407c1c
cd507e407c1c
```

更多的时候，需要让 Docker 在后台运行，而不是直接把执行命令的结果输出在当前宿主机下。此时，可以通过添加“-d”参数来实现。

下面举两个例子来说明一下。如果不使用“-d”参数运行容器，例如：

```
# docker run ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
hello world
hello world
hello world
hello world
```

容器会把输出的结果（STDOUT）打印到宿主机上面。

如果使用了“-d”参数运行容器，例如：

```
# docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
77b2dc01fe0f3f1265df143181e7b9af5e05279a884f4776ee75350ea9d8017a
```

此时，容器会在后台运行，并不会把输出的结果（STDOUT）打印到宿主机上面（输出结果可以用 docker logs 查看）。

容器是否会长久运行，和 docker run 指定的命令有关，和“-d”参数无关。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

使用“-d”参数启动后会返回一个唯一的 ID（镜像后台运行的容器 ID），也可以通过 docker ps 命令来查看容器信息。

```
# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS   NAMES
77b2dc01fe0f   ubuntu:14.04   /bin/sh -c 'while tr     2 minutes ago    Up 1 minute             agitated_wright
```

要获取容器的输出信息，可以通过 docker logs 命令。

```
# docker logs [container ID or NAMES]
hello world
hello world
hello world
...
```

（2）终止容器

可以使用 docker stop 来终止一个运行中的容器。

此外，当 Docker 容器中指定的应用终结时，容器也自动终止。例如，对于之前介绍的内容中只启动了一个终端的容器，用户通过 exit 命令或按 Ctrl+D 组合键来退出终端时，所创建的容器立刻终止。

终止状态的容器可以用 docker ps -a 命令看到，例如：

```
# docker ps -a
CONTAINER ID   IMAGE                COMMAND              CREATED          STATUS                      PORTS   NAMES
ba267838cc1b   ubuntu:14.04         "/bin/bash"          30 minutes ago   Exited （0）                        trusting_newton
                                                                         About a minute ago
98e5efa7d997   training/            "python app.py"      About an         Exited （0）                        backstabbing_
               webapp:latest                             hour ago         34 minutes ago                      pike
```

处于终止状态的容器，可以通过 docker start 命令来重新启动。

此外，docker restart 命令会将一个运行态的容器终止，然后重新启动它。

（3）进入容器

在使用“-d”参数时，容器启动后会进入后台。某些时候需要进入容器进行操作。该操作有很多种方法，包括 docker attach 命令等。下面的示例展示的是如何使用 docker attach 命令。

```
$ sudo docker run -idt ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ sudo docker ps
CONTAINER ID   IMAGE          COMMAND          CREATED          STATUS          PORTS   NAMES
243c32535da7   ubuntu:latest  "/bin/bash"      18 seconds ago   Up 17 seconds           nostalgic_hypatia
$sudo docker attach nostalgic_hypatia
root@243c32535da7:/#
```

注意

使用 attach 命令进入容器需要按 Ctrl+C 组合键。使用 attach 命令有时候并不方便。当多个窗口同时 attach 到同一个容器的时候，所有窗口都会同步显示。当某个窗口因命令阻塞时，其他窗口也无法执行操作了。

版权所有，严禁传播，违者自负法律责任！

（4）导出和导入容器快照

① 导出容器快照。

如果要导出本地某个容器快照，则可以使用 docker export 命令。

```
$ sudo docker ps -a
CONTAINER ID    IMAGE           COMMAND       CREATED          STATUS              PORTS    NAMES
7691a814370e    ubuntu:14.04    "/bin/bash"   36 hours ago     Exited (0)          21 hours ago    test
$ sudo docker export 7691a814370e > ubuntu.tar
```

这样将导出容器快照到本地文件。

② 导入容器快照。

用户可以使用 docker import 从容器快照文件中再导入为镜像，例如：

```
$ cat ubuntu.tar | sudo docker import - test/ubuntu:v1.0
$ sudo docker images
REPOSITORY      TAG     IMAGE ID        CREATED              VIRTUAL SIZE
test/ubuntu     v1.0    9d37a6082e97    About a minute ago   171.3 MB
```

此外，也可以通过指定 URL 或者某个目录来导入，例如：

```
$sudo docker import http://example.com/exampleimage.tgz example/imagerepo
```

注意 》》》》》》

用户既可以使用 docker load 来导入镜像存储文件到本地镜像库，也可以使用 docker import 来导入一个容器快照到本地镜像库。这两者的区别在于，容器快照文件将丢弃所有的历史记录和元数据信息（即仅保存容器当时的快照状态），而镜像存储文件将保存完整记录，体积也大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

（5）删除容器

用户可以使用 docker rm 来删除一个处于终止状态的容器，例如：

```
$sudo docker rm    trusting_newton
trusting_newton
```

如果要删除一个运行中的容器，可以添加“-f”参数。Docker 会发送 SIGKILL 信号给容器。

2. 容器内部操作

① 检查容器主机名。

```
root@1cec78443028:/# hostname
1cec78443028
```

检查容器/etc/hosts 文件。

```
root@1cec78443028:/# cat /etc/hosts
127.0.0.1          localhost
::1        localhost ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2          1cec78443028
```

② 检查容器接口。

```
root@1cec78443028:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
9: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:2/64 scope link
       valid_lft forever preferred_lft forever
```

③ 检查容器的进程。

```
root@1cec78443028:/# ps -aux
USER        PID  %CPU  %MEM    VSZ   RSS  TTY   STAT  START   TIME     COMMAND
root          1   0.0   0.0  18164  2016  ?     Ss    05:25   0:00     bin/bash
root         18   0.0   0.0  15564  1148  ?     R+    05:35   0:00     ps  - aux
```

④ 在第一个容器里安装软件包。

```
root@1cec78443028:/# sudo apt-get install vim
```

3. 查看容器日志及相关操作

① 获取守护式容器的日志。

```
# docker logs ubuntu_container
hello world
hello world
hello world
hello world
```

② 跟踪守护式容器的日志。

```
# docker logs -f ubuntu_container
hello world
hello world
hello world
hello world
```

③ 跟踪守护式容器的最新日志。

```
# docker logs -ft ubuntu_container
2016-12-23T06:08:52.604010000Z hello world
2016-12-23T06:08:53.679457000Z hello world
```

④ 在容器级别启动 syslog。

```
# docker run --log-driver="syslog" --name ubuntu_container -d ubuntu:14.04.3 /bin/sh -c "while true; do echo hello world; sleep 1; done"
441c3de8dbcbfb09d9586c456d47417150dc7cdaf9cfb26597945b27a68bb310
```

⑤ 查看守护式容器的进程。

```
# docker top ubuntu_container
UID     PID     PPID    C    STIME   TTY   TIME       CMD
root    7356    965     0    06:26   ?     00:00:00   /bin/sh -c while true; do echo
                                                      hello world; sleep 1; done
root    7485    7356    0    06:27   ?     00:00:00   sleep 1
```

⑥ 使用 docker stats 命令查看容器占用的系统资源情况。

```
# docker stats
CONTAINER      CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O
441c3de8dbcb   0.10%     2.372 MB / 4.145 GB   0.06%     648 B / 648 B     2.385 MB / 0 B
586478b46dec   3.54%     169.9 MB / 4.145 GB   4.10%     0 B / 0 B         85.86 MB / 3.625 MB
CONTAINER      CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O
441c3de8dbcb   0.10%     2.372 MB / 4.145 GB   0.06%     648 B / 648 B     2.385 MB / 0 B
586478b46dec   3.54%     169.9 MB / 4.145 GB   4.10%     0 B / 0 B         85.86 MB / 3.625 MB
CONTAINER      CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O
441c3de8dbcb   0.09%     2.372 MB / 4.145 GB   0.06%     648 B / 648 B     2.385 MB / 0 B
586478b46dec   3.19%     170 MB / 4.145 GB     4.10%     0 B / 0 B         85.86 MB / 3.625 MB
CONTAINER      CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O
441c3de8dbcb   0.09%     2.372 MB / 4.145 GB   0.06%     648 B / 648 B     2.385 MB / 0 B
586478b46dec   3.19%     170 MB / 4.145 GB     4.10%     0 B / 0 B         85.86 MB / 3.625 MB
CONTAINER      CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O
441c3de8dbcb   0.12%     2.372 MB / 4.145 GB   0.06%     648 B / 648 B     2.385 MB / 0 B
```

⑦ 在容器中运行后台任务。

```
# docker exec -d ubuntu_container touch /etc/new_config_file
```

⑧ 查看容器。

```
# docker inspect ubuntu_container
[
    {
        "Id": "441c3de8dbcbfb09d9586c456d47417150dc7cdaf9cfb26597945b27a68bb310",
        "Created": "2016-12-23T06:25:58.99238638Z",
        "Path": "/bin/sh",
        "Args": [
            "-c",
            "while true; do echo hello world; sleep 1; done"
        ],
...
```

⑨ 查看容器的 IP 地址。

```
# docker inspect --format='{{ .NetworkingSettings.IPAddress }}' ubuntu_container
172.17.0.2
```

项目实训

【实训题目】

从 Docker Hub 上拉一个 Ubuntu 镜像，进行启动、修改、退出等基本操作。

【实训目的】

1. 掌握容器外部操作。
2. 掌握容器内部操作。
3. 掌握容器日志的查看方式及相关信息的查询。

【实训内容】

1. 从 Docker Hub 上下载 Ubuntu 镜像，并使用 docker run 或/bin/bash 运行。
2. 查看 Ubuntu 容器的运行状态。
3. 进入 Ubuntu 容器的内部，查看容器的 hostname、网络等信息。
4. 以多种方式查询容器的日志。

单元小结

本单元介绍了 Docker 镜像的基本使用，读者可了解镜像的相关概念和作用。尤其是在镜像定制和组成结构方面，读者应有更加清晰的认识。对容器的使用和运维方面，读者可加深对容器技术的基本理论的理解，同时可提升实践操作能力。在学习完 Docker 基本的使用方法后，在接下来的单元中将介绍容器的应用服务和集群的管理。

单元 3 Docker 仓库 Registry

学习目标

知识目标

- 了解 Docker 镜像仓库的概念。
- 理解仓库的工作流程。
- 掌握镜像仓库的部署和使用。

技能目标

- 能利用 Docker 构建内部镜像仓库。
- 能通过仓库完成对镜像的管理和使用。

笔 记

学习情境

研发部根据工程师小魏编制的 Docker 容器技术的运维手册在内部推广培训时发现，在对镜像和容器的管理上没有统一规划，容器镜像在线获取和拉取时网络的稳定性不足，延迟较大，并且经常发生超时连接的现象。研发部决定在公司内部建立一个私有仓库，实现公司正常办公和开发的情况下可以随时快速下载和使用容器镜像，摆脱外部条件限制。工程师小魏继续负责该任务的研发和构建，在企业内部建立私有镜像仓库，仔细研究容器的仓库部署、管理和使用方法，进行私有镜像仓库的架构设计。

任务 3.1　Docker 仓库与加速的配置和使用

任务描述

1. 了解 Docker 仓库的几种类别。
2. 利用 Docker 加速器配置 Docker 的镜像源。

知识学习

1. Docker 仓库简介

镜像构建完成后，可以很容易地在当前宿主上运行。但是，如果需要在其他服务器上使用这个镜像，就需要一个集中的存储、分发镜像的服务，Docker Registry（镜像注册）就是这样的服务。一个 Docker Registry 中可以包含多个仓库（Repository）；每个仓库可以包含多个标签（Tag）；每个标签对应一个镜像。

通常，一个仓库会包含同一个软件不同版本的镜像，标签常用于对应该软件的各个版本。开发者可以通过<仓库名>:<标签>的格式来指定具体是这个软件哪个版本的镜像。如果不给出标签，则以 latest 作为默认标签。

以 Ubuntu 镜像为例，Ubuntu 是镜像的名字，其内包含不同的版本标签，如 14.04、16.04。开发者可以通过 ubuntu:14.04 或者 ubuntu:16.04 来具体指定所需哪个版本的镜像。如果忽略了标签，如 Ubuntu，那么将视为 ubuntu:latest。

仓库名经常以两段式路径形式出现，如 xiandian/ubuntu，前者往往是 Docker Registry 多用户环境下的用户名，后者则往往是对应的软件名。但这并非绝对的，取决于所使用的具体 Docker Registry 的软件或服务。

Docker Registry 服务可以分为两种。一种为公开并开放给所有的用户使用，包含用户的搜索、拉取，镜像提交更新，还可以免费保管用户镜像数据。这类服务受制于网络带宽的限制，并不能及时、快速地获取所需要的资源，但是优点是可以获取大部分并可以立即使用的镜像，减少镜像的制作时间。另外一种服务是在一定范围对特定的用户提供 Registry 服务，一般存在于学校内部、企业内部的服务管理和研发等环境，这在一定程度上保证了镜像拉取的速度，对内部的核心镜像数据有保护作用，但是也存在镜像内容不丰富的问题。

（1）公开服务仓库

Docker Registry 公开服务是开放给用户使用，允许用户管理镜像的 Registry 服务。一般，这类公开服务允许用户免费上传、下载公开的镜像，可能提供收费服务，供用户管理私有镜像。最常使用的 Registry 公开服务是官方的 Docker Hub，这也是默认的 Registry，拥有大量高质量的官方镜像。除此以外，还有 CoreOS 的 Quay.io，CoreOS 相关的镜像就存储在这里；Google 的 Google Container Registry，Kubernetes 的镜像使用的就是这个服务。

由于跨洋访问、源站地址不稳定等原因，这些公开服务在国内访问时可能会比较慢。国内的一些云服务商提供了针对 Docker Hub 的镜像服务（Registry Mirror），这些镜像服务被称为加速器。常见的有阿里云加速器、DaoCloud 加速器、灵雀云加速器等。使用加速器可直接从国内的地址下载 Docker Hub 的镜像，比直接从官方网站下载速度会提高很多。后面会进一步讲解如何配置加速器。

国内也有一些云服务商提供类似于 Docker Hub 的公开服务，如时速云镜像仓库、网易云镜像服务、DaoCloud 镜像市场、阿里云镜像库等。

（2）私有服务仓库

除了使用公开服务外，用户还可以在本地搭建私有 Docker Registry。Docker 官方提供了 Docker Registry 镜像，可以直接使用其作为私有 Registry 服务。在后续的相关单元中会进一步讲解如何搭建私有 Registry 服务。

开源的 Docker Registry 镜像只提供了 Docker Registry API 的服务端的实现，但不包含图形界面，以及镜像维护、用户管理、访问控制等高级功能。在官方的商业化版本 Docker Trusted Registry 中，提供了这些高级功能。

2. 镜像注册

下面简单介绍 Docker Hub 的 Docker Registry 的使用。

目前，Docker 官方维护了一个公共仓库 Docker Hub（https://hub.docker.com/），保存的镜像已经超过 15 000 个。对于满足日常需求的大部分镜像，都可以通过在 Docker Hub 中直接下载来实现，目前提供个人私有和公开两种仓库类型的服务。

用户可以通过网页注册。人们只需输入用户名、密码和邮箱就能完成注册和登录，界面如图 3-1-1 所示。

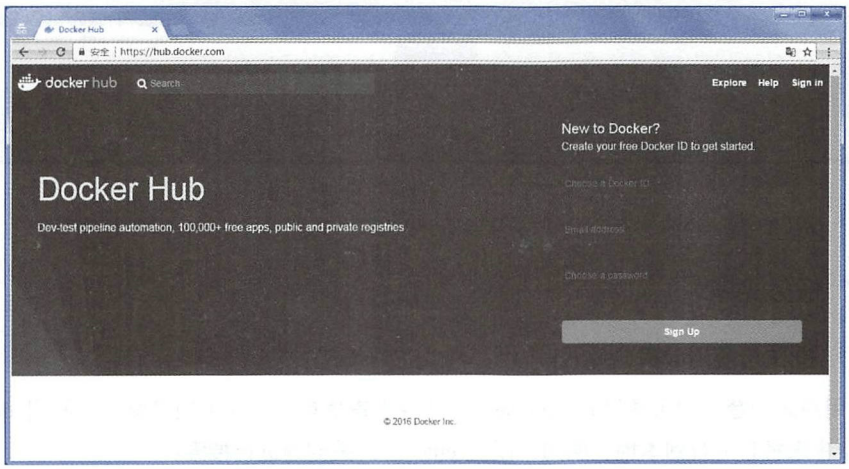


图 3-1-1
Docker Hub 注册和登录界面

注册成功后，进入个人的界面，如图 3-1-2 所示。

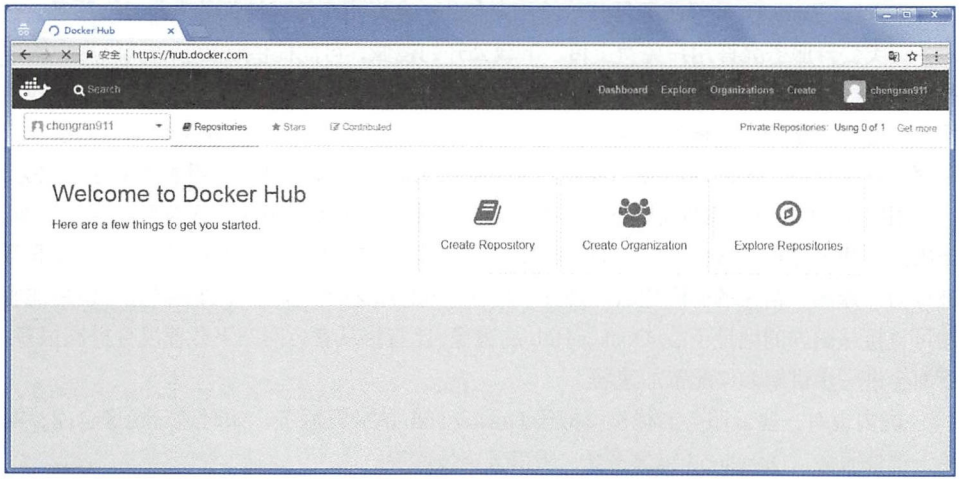


图 3-1-2
Docker Hub 的个人界面

进入个人中心后，用户可以根据自己的需要选择创建公有或私有的镜像，界面如图 3-1-3 所示。

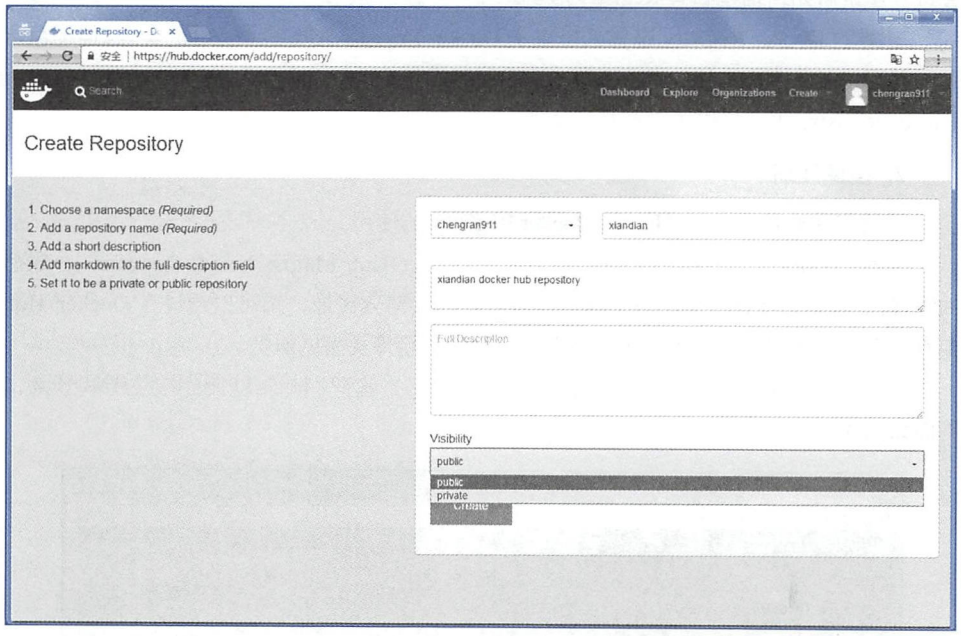


图 3-1-3
Docker Hub 创建仓库界面

任务实施

1. 操作与使用公有仓库

用户无须登录即可通过 docker search 命令来查找官方仓库中的镜像，并利用 docker pull 命令来将它下载到本地。例如，以“centos”为关键词进行搜索：

版权所有，严禁传播，违者自负法律责任！


```
# docker search centos
INDEX      NAME                          DESCRIPTION                    STARS     OFFICIAL  AUTOMATED
docker.io  docker.io/CentOS              The official build             3452                [OK]
                                         of CentOS.
docker.io  docker.io/jdeathe             CentOS-6 6.9 x86_64 /          76                  [OK]
           /CentOS-ssh                   CentOS-7 7.3.1611 x8...
docker.io  docker.io/consol              CentOS container with          27                  [OK]
           /CentOS-xfce-vnc              "headless" VNC sessi...
docker.io  docker.io/nimmis              This is docker images          26                  [OK]
           /java-CentOS                  of CentOS 7 with dif...
docker.io  docker.io/gluster             Official GlusterFS Image       19                  [OK]
           /gluster-CentOS               [ CentOS-7 +  Glu...
docker.io  docker.io/egyptianbman        A simple and highly            9                   [OK]
           /docker-CentOS-nginx-php      configurable docker co...
```

此时可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、星级（表示该镜像的受欢迎程度）、是否官方创建、是否自动创建。官方的镜像说明是官方项目组创建和维护的，AUTOMATED 资源允许用户验证镜像的来源和内容。根据是否是官方提供，可将镜像资源分为两类。一类是类似于 CentOS 这样的基础镜像，称为基础或根镜像。这些基础镜像由 Docker 公司创建、验证、支持和提供。这样的镜像往往使用单个单词作为名字。还有一种类型，如 tianon/CentOS 镜像，它是由 Docker 的用户创建并维护的，往往带有用户名称前缀。可以通过前缀 user_name/来指定使用某个用户提供的镜像，如 tianon 用户。

下载官方“centos”镜像到本地。

```
# docker pull centos
Pulling repository centos
0b443ba03958: Download complete
539c0211cd76: Download complete
511136ea3c5a: Download complete
7064731afe90: Download complete
```

用户也可以在登录后通过 docker push 命令来将镜像推送到 Docker Hub。

2. 配置与使用加速器

在国内访问 Docker Hub 有时会遇到困难，此时可以配置镜像加速器。国内的很多云服务商都提供了加速器服务，例如阿里云加速器、DaoCloud 加速器、灵雀云加速器。

使用 DaoCloud 获取的加速脚本修改本机中的 Docker 的镜像地址，如下所示。

```
# curl -sSL  https://get.daocloud.io/daotools/set_mirror.sh | sh -s  http://ef0cb1d0.m.daocloud.io
Success.
You need to restart docker to take effect: sudo systemctl restart docker
# systemctl restart docker
```

项目实训

【实训题目】

登录 Docker Hub 网站，注册一个用户，在 Docker Hub 上创建一个公有的镜像仓库，

版权所有，严禁传播，违者自负法律责任！

使用 DaoCloud 加速器取代默认的 Docker 的镜像源。

笔 记

【实训目的】

1. 掌握公有仓库的使用方法和流程。
2. 掌握第三方的加速器的配置和使用。

【实训内容】

1. 登录 Docker Hub 网站，注册一个用户。
2. 通过 Docker Hub 创建一个私有仓库。
3. 使用 docker search 命令实现镜像搜索。
4. 使用 docker pull 命令实现镜像的拉取。

任务 3.2　Docker 私有仓库与管理

任务描述

1. 了解 Docker 私有仓库的特点和运行流程。
2. 配置 Docker 私有仓库，为后续服务提供仓库源。
3. 整理过程步骤，形成规范的操作、运维、实施手册。

知识学习

1. 私有仓库特点

仓库（Repository）是集中存放镜像的地方，在上个任务中读者知道了 Docker 仓库可以分为公有仓库和私有仓库，然而公有仓库在某些情况下并不适用于公司内部传输。通过对比两种仓库的特点，大致可以得出私有仓库有以下优点。

① 省带宽。

② 传输速度快。

③ 方便存储。

2. Docker Registry 工作方式

Docker Registry 是 Image 的仓库，当开发者编译完成一个 Image 时，就可以推送到公共的 Registry，如 Docker Hub，也可以推送到自己的私有 Registry。使用 Docker Client，开发者可以搜索已经发布的 Image，从中拉取 Image 到本地，并在容器中运行。

Docker Hub 提供了公有和私有的 Registry。所有人都可以搜索和下载公共镜像。私有仓库只有私有用户才能查询和下载。

3. 库配置文件

Docker 的 Registry 利用配置文件提供了一些仓库的模板（Flavor），用户可以直接使用它们来进行开发或生产部署。

在 config_sample.yml 文件中，可以看到一些现成的模板段：

```
common：基础配置。
```

版权所有，严禁传播，违者自负法律责任！


```
local：存储数据到本地文件系统。
s3：存储数据到 AWS S3 中。
dev：使用 local 模板的基本配置。
test：单元测试时使用。
prod：生产环境配置（基本上跟 s3 配置类似）。
gcs：存储数据到 Google 的云存储。
swift：存储数据到 OpenStack Swift 服务。
glance：存储数据到 OpenStack Glance 服务，本地文件系统为后备。
glance-swift：存储数据到 OpenStack Glance 服务，Swift 为后备。
elliptics：存储数据到 Elliptics key/value。
```

用户也可以添加自定义的模板段。默认情况下使用的模板是 dev，要使用某个模板作为默认值，可以添加 SETTINGS_FLAVOR 到环境变量中，例如：

```
export SETTINGS_FLAVOR=dev
```

另外，配置文件中支持从环境变量中加载值，语法格式为：

```
_env:VARIABLENAME[:DEFAULT]。
```

示例配置：

```
common:
loglevel: info
search_backend: "_env:SEARCH_BACKEND:"
sqlalchemy_index_database:
"_env:SQLALCHEMY_INDEX_DATABASE:sqlite:////tmp/docker-registry.db"

prod:
loglevel: warn
storage: s3
s3_access_key: _env:AWS_S3_ACCESS_KEY
s3_secret_key: _env:AWS_S3_SECRET_KEY
s3_bucket: _env:AWS_S3_BUCKET
boto_bucket: _env:AWS_S3_BUCKET
storage_path: /srv/docker
smtp_host: localhost
from_addr: docker@myself.com
to_addr: my@myself.com

dev:
loglevel: debug
storage: local
storage_path: /home/myself/docker

test:
storage: local
storage_path: /tmp/tmpdockertmp
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

任务实施

1. 准备环境

操作系统为 CentOS 7.1，内核大版本为 3.10，Docker 版本为 1.10.3。将需要的软件包 PAAS-XianDian-v2.0.iso 复制到系统中。

2. 部署与使用 Docker 镜像仓库

导入并运行 Registry 镜像。

```
# docker load < registry_latest.tar
# docker run -d -p 5000:5000 --restart=always --name registry registry:latest
```

（1）在本机查看已有的镜像

```
# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker.io/registry  latest              c9bd19d022f6        8 months ago        33.27 MB
```

（2）上传本地资源到私有仓库

① 修改默认监听端口。

对于 Docker 仓库节点，修改 Docker 默认进程绑定端口为 2375 端口，编辑配置文件 /etc/sysconfig/docker，在 OPTIONS 配置项后添加以下内容：-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock。完成后重启 Docker 服务。

② 上传 Nginx 容器到 Docker 镜像仓库中。

```
# docker load < nginx_latest.tar
# docker tag 548cb73bef9a   10.0.6.83:5000/nginx:latest
# docker push     10.0.6.83:5000/nginx:latest
The push refers to a repository [localhost:5000/nginx]
0b297cc91255: Pushed
cbb475ff5c8e: Pushed
2eea2d5e43e6: Pushed
b6ca02dfe5e6: Pushed
latest: digest: sha256:6553bb0f5057f7414caca83f05add6866d71fb22ba4655a1298a013e1d8be
fe1 size: 1138
```

（3）下载私有仓库的镜像

① 修改镜像地址。

对于客户端节点，首先需要修改本机的 Docker 配置文件/etc/sysconfig/docker，修改并添加仓库地址，修改配置项 ADD_REGISTRY='--add-registry 10.0.6.83:5000'，另外修改配置 INSECURE_REGISTRY='--insecure-registry 10.0.6.83:5000'。完成后重启 Docker 服务。

② 获取仓库镜像。

```
# docker pull 10.0.6.83:5000/nginx:latest
Trying to pull repository 10.0.6.83:5000/nginx
latest: Pulling from 10.0.6.83:5000/nginx
386a066cd84a: Pull complete
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
524267bc200a: Pull complete
476d61c7c43a: Pull complete
0750d0e28b90: Pull complete
c2a78c5fad8e: Pull complete
1cd7ed6205ba: Pull complete
35f9221ee11f: Pull complete
708e45912fb3: Pull complete
deca6f1cf04d: Pull complete
Digest: sha256:28feee8c0e5c19325bf09cfa489cd30e1c3a1e574dae8b6c2fb7f468463b6b5c
Status: Downloaded newer image for 10.0.6.83:5000/nginx:latest
```

（4）用 curl 查看仓库中的镜像

```
# curl http://127.0.0.1:2375/images/json | python -m json.tool
    {
        "Created": 1492739802,
        "Id": "sha256:548cb73bef9af70e4bc2874efa7af1a4b73e127f03b3c68338e460082a53209e",
        "Labels": {},
        "ParentId": "",
        "RepoDigests": null,
        "RepoTags": [
            "localhost:5000/nginx:latest"
        ],
        "Size": 231251987,
        "VirtualSize": 231251987
    },
    {
        "Created": 1476835847,
        "Id": "sha256:c9bd19d022f6613fa0e3d73b2fe2b2cffe19ed629a426db9a652b597fccf07d4",
        "Labels": {},
        "ParentId": "",
        "RepoDigests": null,
        "RepoTags": [
            "docker.io/registry:latest"
        ],
        "Size": 33274815,
        "VirtualSize": 33274815
    }
]
```

（5）批量上传镜像

如果要上传其他的镜像到私有仓库内，可以使用提供的 install.sh 脚本进行。

执行结果如下：

```
# ./install.sh
Please wait, checking the network.....
Check completed
/opt/docker_images /opt
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
7aa9047e618a8857fbeac4de2eccc639486e70507ca94f77443653f838d62653
please wait ...
The push refers to a repository [10.0.6.83:5000/registry]
…
7fa4fac65171: Verifying Checksum
7fa4fac65171: Download complete
7fa4fac65171: Pull complete
7fa4fac65171: Pull complete
33de63de5fdb: Pull complete
33de63de5fdb: Pull complete
d00b3b942272: Pull complete
d00b3b942272: Pull complete
Digest: sha256:9675a74127c4fc5833b7aba45e41e26f6385e330d72a784f43cddac81c293eab
Status: Downloaded newer image for docker.io/rancher/agent:v1.0.2
```

项目实训

【实训题目】

安装 Docker 镜像仓库，并将本地的镜像文件上传到仓库中，在其他节点拉取镜像来测试仓库是否正常运行。

【实训目的】

1. 掌握私有仓库的概念和使用方法。
2. 掌握私有仓库的部署流程和配置方法。

【实训内容】

1. 利用提供的镜像构建私有仓库。
2. 上传提供的镜像到仓库中。
3. 使用 Docker 的相关命令从私有仓库拉取上传镜像。

单元小结

本单元首先通过 Docker 的两种不同类型的镜像仓库进行讲解分析，介绍各自的相同点和差异点，并且深入介绍了私有仓库的使用范围和使用方式。在构建私有仓库部分，通过一个简单的镜像仓库构建的过程，读者可了解其中的搭建流程，掌握部署的要点，加深对仓库的认识。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

单元 4 Docker 网络、存储和接口

学习目标

知识目标

- 了解 Docker 存储技术的运用。
- 了解 Docker 网络的配置和使用。
- 了解 Docker API 的使用。
- 了解 Docker 的底层技术（命名空间、控制组、联合文件系统和容器管理）。

技能目标

- 掌握 Docker 存储的数据卷，可以使用数据卷对数据进行备份、恢复和迁移。
- 掌握 Docker 网络的配置、容器互联、配置网桥等操作。
- 掌握 Docker Remote API 的使用，可以通过 API 来管理 Docker 镜像和容器。
- 掌握 Docker 底层技术，会使用命名空间 NameSpace，会管理与使用控制组 Cgroups，会对联合文件系统 UnionFS 和容器进行管理。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

笔 记

学习情境

经过这段时间对 Docker 容器和镜像的使用，研发部有很多组员在使用过程中不太明白底层技术的使用原理，对容器也只是停留在简单使用和维护上。于是继续安排工程师小魏深入研究 Docker 底层技术，对 Docker 底层技术进行测试。对于高级运维，希望能从容器的存储、网络、API 等几个方面进行研究整理。小魏接到任务后，积极查阅相关资料，编写理论文档，构建实验验证。

任务 4.1　Docker 存储

任务描述

1. 了解 Docker 的存储方式，包括数据卷和数据卷容器。
2. 会对数据卷进行创建、删除、挂载操作。
3. 会使用数据卷容器来备份、恢复、迁移数据卷。

知识学习

1. 数据卷

数据卷（Data Volumes）是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多如下有用的特性。

① 数据卷可以在容器之间共享和重用。

② 对数据卷的修改会立马生效。

③ 对数据卷的更新，不会影响镜像。

④ 数据卷默认会一直存在，即使容器被删除。

数据卷的使用，类似于 Linux 下对目录或文件进行 mount 操作，镜像中被指定为挂载点的目录中的文件会被隐藏掉，能显示的是挂载的数据卷。

2. 数据卷容器

数据卷容器（Data Volume Containers）涉及容器间共享的持久化、序列化的数据。数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其他容器挂载的。

任务实施

（1）创建一个数据卷

在用 docker run 命令的时候，可使用“-v”标记来创建一个数据卷并挂载到容器里。在一次 run 中多次使用可以挂载多个数据卷。下面创建一个名为 web 的容器，并加载一个数据卷到容器的“/webapp”目录。

```
# sudo docker run -d -P --name web -v /webapp training/webapp python app.py
```


非卖品，仅供非商业用途或交流学习使用

可以在Dockerfile中使用VOLUME 来添加一个或者多个新的卷到由该镜像创建的任意容器。

（2）删除数据卷

数据卷是被用来持久化数据的，它的生命周期独立于容器。Docker 不会在容器被删除后自动删除数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的数据卷。如果需要在删除容器的同时移除数据卷，则可以在删除容器的时候使用 docker rm -v 这个命令。

（3）挂载一个主机目录作为数据卷

使用“-v”标记也可以指定挂载一个本地主机的目录到容器中去。

```
# docker run -d -P --name web -v /src/webapp:/opt/webapp training/webapp python app.py
```

上面的命令加载主机的“/src/webapp”目录到容器的“/opt/webapp”目录。这个功能在进行测试的时候十分方便，比如用户可以放置一些程序到本地目录中，来查看容器是否正常工作。本地目录的路径必须是绝对路径，如果目录不存在，Docker 会自动为开发者创建。

Docker 挂载数据卷的默认权限是读写，用户也可以通过“:ro”指定为只读。

```
# docker run -d -P --name web -v /src/webapp:/opt/webapp:ro
training/webapp python app.py
```

（4）查看数据卷的具体信息

在主机里使用以下命令可以查看指定容器的信息，在输出的内容中找到其中和数据卷相关的部分，可以看到所有的数据卷都是创建在主机的“/var/lib/docker/volumes/”下面的。

```
# docker inspect web
...
"Volumes": {
    "/webapp": "/var/lib/docker/volumes/fac362...80535"
},
"VolumesRW": {
    "/webapp": true
}
...
```

（5）挂载一个本地主机文件作为数据卷

“-v”标记也可以从主机挂载单个文件到容器中。

```
# docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu /bin/bash
```

这样就可以记录在容器输入过的命令了。如果直接挂载一个文件，则很多文件编辑工具，包括 vi 或者 sed --in-place，都可能会造成文件 inode 的改变，所以最简单的办法就是直接挂载文件的父目录。

① 创建数据卷容器。

创建一个名为 dbdata 的数据卷容器：

非卖品，仅供非商业用途或交流学习使用

```
# docker run -d -v /dbdata --name dbdata training/postgres echo Data-only container for postgres
```

在其他容器中使用--volumes-from 来挂载 dbdata 容器中的数据卷。

```
# docker run -d --volumes-from dbdata --name db1 training/postgres
# docker run -d --volumes-from dbdata --name db2 training/postgres
```

用户可以使用多于一个的--volumes-from 参数来指定从多个容器挂载不同的数据卷。也可以从其他已经挂载了数据卷的容器来级联挂载数据卷。

```
# docker run -d --name db3 --volumes-from db1 training/postgres
```

注意

使用--volumes-from 参数所挂载数据卷的容器并不需要保持在运行状态。

如果删除了挂载的容器（包括 dbdata、db1 和 db2），那么数据卷并不会被自动删除。如果要删除一个数据卷，则必须在删除最后一个还挂载着它的容器时使用 docker rm -v 命令来指定同时删除关联的容器，这可以让用户在容器之间升级和移动数据卷。

② 利用数据卷容器来备份、恢复、迁移数据卷。

用户可以利用数据卷对其中的数据进行备份、恢复和迁移。

- 备份。

首先使用--volumes-from 标记来创建一个加载 dbdata 容器卷的容器，并从主机挂载当前目录到容器的“/backup”目录，命令如下：

```
# docker run --volumes-from dbdata -v $（pwd）:/backup ubuntu tar cvf /backup/backup.tar /dbdata
```

容器启动后，使用了 tar 命令来将 dbdata 卷备份为容器中的/backup/backup.tar 文件，也就是主机当前目录下的名为 backup.tar 的文件。

- 恢复。

如果要恢复数据到一个容器，首先创建一个带有空数据卷的容器 dbdata2。

```
# docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

然后创建另一个容器来挂载 dbdata2 容器卷中的数据卷，并使用 untar 解压备份文件到挂载的容器卷中。

```
# docker run --volumes-from dbdata2 -v $（pwd）:/backup busybox tar xvf
/backup/backup.tar
# untar/backup/backup.tar
```

为了查看/验证恢复的数据，可以再启动一个容器挂载同样的容器卷。

```
# docker run --volumes-from dbdata2 busybox /bin/ls /dbdata
```

项目实训

【实训题目】

创建一个名字为 webapp 的数据卷，并查看数据卷的详细信息。使用数据卷容器（dbdata）对数据卷进行备份、恢复操作。

非卖品，仅供非商业用途或交流学习使用

笔记

【实训目的】

1. 掌握数据卷的创建、删除和挂载使用。
2. 掌握数据卷容器的创建方法。
3. 掌握数据卷容器的备份和恢复操作。

【实训内容】

1. 使用命令创建一个数据卷，创建一个名为 web 的容器，并加载一个数据卷到容器的“/webapp”目录。

2. 在主机里使用命令 docker inspect web 可以查看指定容器的信息，在输出的内容中找到其中和数据卷相关的部分。

3. 创建一个名为 dbdata 的数据卷容器。

4. 备份：首先使用--volumes-from 标记来创建一个加载 dbdata 容器卷的容器，并从主机挂载当前目录到容器的“/backup”目录。容器启动后，使用 tar 命令来将 dbdata 卷备份为容器中的/backup/backup.tar 文件，也就是主机当前目录下的名为 backup.tar 的文件。

5. 恢复：如果要恢复数据到一个容器，则首先需要创建一个带有空数据卷的容器 dbdata2，然后创建另一个容器来挂载 dbdata2 容器卷中的数据卷，并使用 untar 解压备份文件到挂载的容器卷中。

任务 4.2　Docker 网络

任务描述

1. 了解 Docker 容器网络，了解如何配置基础网络，包括如何创建网络命名空间、绑定接口到网桥、端口映射等。

2. 了解 Docker 高级网络配置，包括配置 DNS，容器访问控制，映射端口，自定义网桥等。

知识学习

1. Docker 网络功能介绍

Docker 的网络实现其实就是利用 Linux 上的网络命名空间、网桥和虚拟网络设备（VETH）等来实现。默认情况下，Docker 安装完成后会创建一个网桥 docker0。Docker 中的网络接口默认都是虚拟的网络接口。Docker 容器网络在本地主机和容器内分别创建一个虚拟接口，并让它们彼此连通。

Docker 创建一个容器需执行如下操作。

① 创建一对虚拟接口，分别放到本地主机和新容器中。

② 本地主机一端桥接到默认的 docker0 或指定网桥上，并具有一个唯一的名字，如 veth65f9。

③ 容器一端放到新容器中，并修改名称为 eth0，这个接口只在容器的命名空间可见。

④ 从网桥可用地址段中获取一个空闲地址 IP，分配给容器的 eth0，并配置默认路由

非卖品，仅供非商业用途或交流学习使用

到桥接网卡 veth65f9。

⑤ 完成这些之后，容器就可以使用 eth0 虚拟网卡来连接其他容器和其他网络。

Docker 容器网络示意图如图 4-2-1 所示。另外，可以在使用 docker run 的时候通过--net 参数来指定容器的网络配置，有以下 4 个可选值。

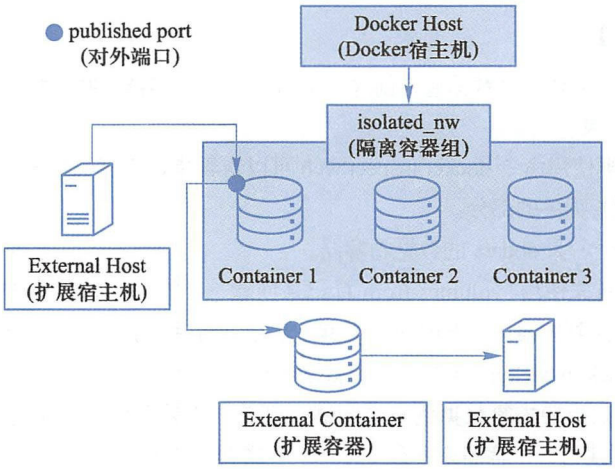


图 4-2-1
Docker 容器网络示意图

--net=bridge：这个是默认值，连接到默认的网桥。

--net=host：告诉 Docker 不要将容器网络放到隔离的命名空间中，使用本地主机的网络，它拥有完全的本地主机接口访问权限。容器进程可以跟主机的其他 root 进程一样打开低范围的端口机。因此使用这个选项的时候要非常小心。如果进一步使用--privileged=true，则容器会被允许直接配置主机的网络堆栈。

--net=container:NAME_or_ID：让 Docker 将新建容器的进程放到一个已存在容器的网络栈中，新容器进程有自己的文件系统、进程列表和资源限制，但会和已存在的容器共享 IP 地址和端口等网络资源，两者的进程可以直接通过 lo 环回接口通信。

--net=none：让 Docker 将新容器放到隔离的网络栈中，但是不进行网络配置。之后，用户可以自己进行配置。

当容器结束后，Docker 会清空容器，容器内的 eth0 会随网络命名空间一起被清除，A 接口也自动从 docker0 卸载。

2. Docker 高级网络配置

这里将介绍 Docker 的一些高级网络配置和选项。

当 Docker 启动时，会自动在主机上创建一个 docker0 虚拟网桥，实际上是 Linux 的一个 bridge，可以理解为一个软件交换机。它会在挂载到它的接口之间进行转发。

同时，Docker 随机分配一个本地未占用的私有网段（在RFC1918中定义）中的一个地址给 docker0 接口。例如典型的 172.17.42.1，掩码为 255.255.0.0。此后启动的容器内的接口也会自动分配一个同一网段（172.17.0.0/16）的地址。

当创建一个 Docker 容器的时候，同时会创建一对 veth pair 接口（当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包）。这对接口一端在容器内，即 eth0；另一端在本地，并被挂载到 docker0 网桥，名称以 veth 开头（例如 vethAQI2QT）。通过这种方式，主机可以跟容器通信，容器之间也可以相互通信。此时，Docker 就创建了主机和

非卖品，仅供非商业用途或交流学习使用

所有容器之间的一个虚拟共享网络，如图 4-2-2 所示。

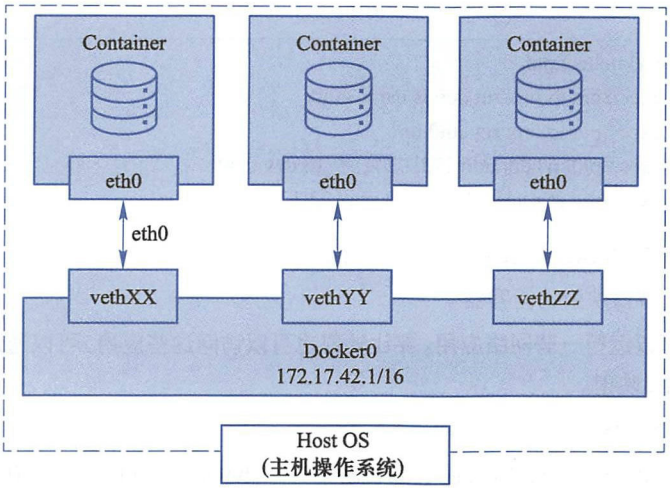


图 4-2-2
Docker 网络

接下来的部分将介绍一些场景中的 Docker 所有的网络定制配置，以及通过 Linux 命令来调整、补充甚至替换 Docker 默认的网络配置。

任务实施

(1) 配置容器网络

用户可以使用--net=none 参数，也可以自行配置网络，让容器具有访问网络的权限。通过这个过程，用户可以了解 Docker 配置网络的细节。

① 启动一个/bin/bash 容器，指定--net=none 参数。

```
# docker run -i -t --rm --net=none base /bin/bash
root@63f36fc01b5f:/#
```

② 在本地主机查找容器的进程 ID，并为它创建网络命名空间。

```
# docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
# mkdir -p /var/run/netns
# ln -s /proc/$pid/ns/net /var/run/netns/$pid
```

③ 检查桥接网卡的 IP 和子网掩码信息。

```
# ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...
```

④ 创建一对 veth pair 接口 A 和 B，绑定 A 到网桥 docker0，并启用它。

```
# ip link add A type veth peer name B
# brctl addif docker0 A
# ip link set A up
```

⑤ 将 B 放到容器的网络命名空间，命名为 eth0，启动它并配置一个可用 IP（桥接网段）和默认网关。

```
# ip link set B netns $pid
# ip netns exec $pid ip link set dev B name eth0
# ip netns exec $pid ip link set eth0 up
# ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0
# ip netns exec $pid ip route add default via 172.17.42.1
```

以上，就是 Docker 配置网络的具体过程。

（2）通过端口映射访问容器

容器中可以运行一些网络应用。要让外部也可以访问这些应用，可以通过“-P”或“-p”参数来指定端口映射。

1）查看端口信息

当使用“-P”标记时，Docker 会随机映射一个 49000 ~ 49900 的端口到内部容器开放的网络端口。

使用 docker ps 可以看到，本地主机的 49155 端口被映射到了容器的 5000 端口。此时访问本机的 49155 端口即可访问容器内 Web 应用提供的界面。

```
# docker run -d -P training/webapp python app.py
# docker ps -l
CONTAINER ID  IMAGE                COMMAND       CREATED        STATUS         PORTS                    NAMES
bc533791f3f5  training/webapp:latest  python app.py  5 seconds ago  Up 2 seconds  0.0.0.0:49155 -> 5000/tcp  nostalgic_morse
```

同样的，可以通过 docker logs 命令来查看应用的信息。

```
# docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -
```

使用“-p”标记，则可以指定要映射的端口，并且在一个指定端口上只可以绑定一个容器。支持的格式如下：

```
ip:hostPort:containerPort | ip::containerPort | hostPort:containerPort。
```

2）映射所有接口地址

使用 hostPort:containerPort 格式将本地的 5000 端口映射到容器的 5000 端口，可以执行：

```
# docker run -d -p 5000:5000 training/webapp python app.py
```

此时默认会绑定本地所有接口上的所有地址。

3）映射到指定地址的指定端口

用户可以使用 ip:hostPort:containerPort 格式指定映射使用一个特定地址，比如 localhost 地址 127.0.0.1。

```
# docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py
```

4）映射到指定地址的任意端口

使用 ip::containerPort 格式绑定 localhost 的任意端口到容器的 5000 端口，本地主机会自动分配一个端口。

```
# docker run -d -p 127.0.0.1::5000 training/webapp python app.py
```

还可以使用 udp 标记来指定 udp 端口。

```
# docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

5）查看映射端口配置

使用 docker port 可查看当前映射的端口配置，也可以查看到绑定的地址。

```
# docker port nostalgic_morse 5000
127.0.0.1:49155.
```

容器有自己的内部网络和 IP 地址，可以多次使用“-p”标记来绑定多个端口，例如：

```
# sudo docker run -d -p 5000:5000   -p 3000:80 training/webapp python app.py
```

（3）配置容器互联

容器的连接（Linking）系统是除了端口映射外，另一种跟容器中的应用交互的方式。该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

1）自定义容器名称

连接系统依据容器的名称来执行。因此，首先需要自定义一个好记的容器名称。使用--name 标记可以为容器自定义名。

```
# sudo docker run -d -P --name web training/webapp python app.py
```

使用 docker ps 来验证设定的名称。

```
# docker ps -l
CONTAINER ID  IMAGE             COMMAND       CREATED       STATUS        PORTS                    NAMES
aed84ee21bde  training/         python app.py 12 hours ago  Up 2 seconds  0.0.0.0:49154            web
              webapp:latest                                                -> 5000/tcp
```

也可以使用 docker inspect 来查看容器的名称。

```
# docker inspect -f "{{ .Name }}" aed84ee21bde
/web
```

注意

容器的名称是唯一的。如果已经命名了一个 web 的容器，当开发者要再次使用 web 这个名称的时候，需要先用 docker rm 来删除之前创建的同名容器。

在执行 docker run 的时候，如果添加--rm 标记，则容器在终止后会立刻删除。

注意

--rm 和-d 参数不能同时使用。

2）容器互联

使用--link 参数可以让容器之间安全地进行交互。

下面先创建一个新的数据库容器。

```
$ sudo docker run -d --name db training/postgres
```

删除之前创建的 web 容器。

```
$ docker rm -f web
```

然后创建一个新的 web 容器，并将它连接到 db 容器。

```
$ sudo docker run -d -P --name web --link db:db training/webapp python app.py
```

此时，db 容器和 web 容器建立了互联关系。

--link 参数的格式为--link name:alias，其中，name 是要连接的容器的名称，alias 是这个连接的别名。

使用 docker ps 来查看容器的连接。

```
# docker ps
CONTAINER ID  IMAGE                COMMAND                CREATED         STATUS         PORTS                    NAMES
349169744e49  training/            su postgres            About a         Up About       5432/tcp                 db, web/db
              postgres:latest      -c '/usr               minute ago      a minute
aed84ee21bde  training/            python                 16 hours ago    Up 2 minutes   0.0.0.0:49154-           web
              webapp:latest        app.py                                                >5000/tcp
```

此时可以看到自定义名称的容器 db 和 web，db 容器的 names 列有 db 也有 web/db。这表示 web 容器连接到 db 容器，web 容器将被允许访问 db 容器的信息。

Docker 在两个互联的容器之间创建了一个安全隧道，而且不用映射它们的端口到宿主机上。在启动 db 容器的时候并没有使用“-p”和“-P”标记，从而避免了暴露数据库端口到外部网络上。

使用 env 命令来查看 web 容器的环境变量。

```
# docker run --rm --name web2 --link db:db training/webapp env
...
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5000_TCP=tcp://172.17.0.5:5432
DB_PORT_5000_TCP_PROTO=tcp
DB_PORT_5000_TCP_PORT=5432
DB_PORT_5000_TCP_ADDR=172.17.0.5
...
```

其中，以 DB_开头的环境变量是供 web 容器连接 db 容器使用的，前缀采用大写的连接别名。

除了环境变量外，Docker 还添加 host 信息到父容器的/etc/hosts 的文件。下面是父容器 web 的 hosts 文件。

这里有两个 hosts，第一个是 web 容器，web 容器用 ID 作为其主机名，第二个是 db 容器的 IP 和主机名。用户可以在 web 容器中使用 ping 命令来测试与 db 容器的连通。

```
# docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
...
172.17.0.5  db
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db
PING db （172.17.0.5）: 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

用 ping 来测试 db 容器，它会解析成 172.17.0.5。

注意

官方的 Ubuntu 镜像默认没有安装 ping，需要自行安装。

用户可以连接多个父容器到子容器，如可以连接多个 web 到 db 容器上。

（4）配置高级网络

1）命令列表

下面是一个跟 Docker 网络相关的命令列表。其中，有些命令选项只有在 Docker 服务启动的时候才能配置，而且不能马上生效。

```
-b BRIDGE or --bridge=BRIDGE --指定容器挂载的网桥。
--bip=CIDR --定制 docker0 的掩码。
-H SOCKET... or --host=SOCKET... --Docker 服务端接收命令的通道。
--icc=true|false --是否支持容器之间进行通信。
--ip-forward=true|false –是否开启 IPV4 转发的功能。
--iptables=true|false --是否允许 Docker 添加 iptables 规则。
--mtu=BYTES --容器网络中的 MTU。
```

下面两个命令选项既可以在启动服务时指定，也可以在 Docker 容器启动（docker run）的时候指定。在 Docker 服务启动的时候指定则会成为默认值，后面执行 docker run 时可以覆盖设置的默认值。

```
--dns=IP_ADDRESS... --使用指定的 DNS 服务器。
--dns-search=DOMAIN... --指定 DNS 搜索域。
```

最后这些选项只有在 docker run 执行时使用，因为它是针对容器的特性内容的。

```
-h HOSTNAME or --hostname=HOSTNAME --配置容器主机名。
--link=CONTAINER_NAME:ALIAS --添加到另一个容器的连接。
--net=bridge|none|container:NAME_or_ID|host --配置容器的桥接模式。
```

```
-p SPEC or --publish=SPEC --映射容器端口到宿主机。
-P or --publish-all=true|false --映射容器所有端口到宿主机。
```

2）配置 DNS

Docker 没有为每个容器专门定制镜像，那么怎么自定义配置容器的主机名和 DNS 配置呢？秘诀就是它利用虚拟文件来挂载到容器的 3 个相关配置文件。

在容器中使用 mount 命令可以看到挂载信息：

```
# mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
tmpfs on /etc/resolv.conf type tmpfs ...
...
```

这种机制可以在宿主机 DNS 信息更新后，所有 Docker 容器的 DNS 配置通过/etc/resolv.conf 文件立刻得到更新。

如果用户想要手动指定容器的配置，则可以利用下面的选项。

- -h HOSTNAME or --hostname=HOSTNAME：设定容器的主机名，它会被写到容器内的/etc/hostname 和/etc/hosts。但它在容器外部看不到，既不会在 docker ps 中显示，也不会在其他容器的/etc/hosts 看到。
- --link=CONTAINER_NAME:ALIAS：该选项会在创建容器的时候添加一个其他容器的主机名到/etc/hosts 文件中，让新容器的进程使用主机名 ALIAS 就可以连接它。
- --dns=IP_ADDRESS：添加 DNS 服务器到容器的/etc/resolv.conf 中，让容器用这个服务器来解析所有不在/etc/hosts 中的主机名。
- --dns-search=DOMAIN：设定容器的搜索域，当设定搜索域为.example.com 时，在搜索一个名为 host 的主机时，DNS 不仅搜索 host，还会搜索 host.example.com。

注意

如果没有上述最后两个选项，Docker 会默认用主机上的/etc/resolv.conf 来配置容器。

3）容器访问控制

容器的访问控制，主要通过 Linux 上的 iptables 防火墙来进行管理和实现。iptables 是 Linux 上默认的防火墙软件，在大部分发行版中都自带。

① 容器访问外部网络。

容器要想访问外部网络，需要本地系统的转发支持。在 Linux 系统中，检查转发是否打开：

```
$sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

如果为 0，说明没有开启转发，则需要手动打开。

```
$sysctl -w net.ipv4.ip_forward=1
```

如果在启动 Docker 服务的时候设定--ip-forward=true，Docker 就会自动设定系统的

ip_forward 参数为 1。

② 容器之间的访问。

容器之间相互访问，需要两方面的支持。

一是容器的网络拓扑是否已经互联。默认情况下，所有的容器都会被连接到 docker0 网桥上。

二是本地系统的防火墙软件 iptables 是否允许通过。

③ 访问所有端口。

当启动 Docker 服务的时候，默认会添加一条转发策略到 iptables 的 FORWARD 链上。策略是通过（ACCEPT）还是禁止（DROP），取决于配置是--icc=true（默认值）还是--icc=false。当然，如果手动指定--iptables=false，则不会添加 iptables 规则。

可见，默认情况下，不同容器之间是允许网络互通的。如果为了安全考虑，则可以在/etc/default/docker 文件中通过配置 DOCKER_OPTS=--icc=false 来禁止它。

④ 访问指定端口。

通过-icc=false 关闭网络访问后，还可以通过--link=CONTAINER_NAME:ALIAS 选项来访问容器的开放端口。

例如，在启动 Docker 服务时，可以同时使用 icc=false 和--iptables=true 参数来关闭允许相互的网络访问，并让 Docker 可以修改系统中的 iptables 规则。

此时，系统中的 iptables 规则可能如下：

```
$ sudo iptables -nL
...
Chain FORWARD （policy ACCEPT）
target          prot opt source          destination
DROP            all  --  0.0.0.0/0       0.0.0.0/0
...
```

之后，启动容器（docker run）时使用--link=CONTAINER_NAME:ALIAS 选项，Docker 会在 iptables 中为两个容器分别添加一条 ACCEPT 规则，允许相互访问开放的端口（取决于 Dockerfile 中的 EXPOSE 行）。

当添加了--link=CONTAINER_NAME:ALIAS 选项后，便添加了 iptables 规则。

```
$ sudo iptables -nL
...
Chain FORWARD （policy ACCEPT）
target          prot opt source          destination
ACCEPT          tcp  --  172.17.0.2      172.17.0.3   tcp spt:80
ACCEPT          tcp  --  172.17.0.3      172.17.0.2   tcp dpt:80
DROP            all  --  0.0.0.0/0       0.0.0.0/0
```

注意

当使用--link=CONTAINER_NAME:ALIAS 选项时，参数 CONTAINER_NAME 必须是启动容器时分配的名字或使用--name 参数指定的名字。使用其他的容器，主机名则不会被识别。

版权所有，严禁传播，违者自负法律责任！

4）映射容器端口到宿主机的实现

默认情况下，容器可以主动访问到外部网络的连接，但是外部网络无法访问到容器。

① 容器访问外部实现。

容器所有到外部网络的连接，源地址都会被伪装成本地系统的 IP 地址。这是使用 iptables 的源地址伪装操作实现的。

查看主机的 NAT 规则。

```
# iptables -t nat -nL
...
Chain POSTROUTING （policy ACCEPT）
target          prot opt source          destination
MASQUERADE      all  --  172.17.0.0/16    !172.17.0.0/16
...
```

其中，上述规则将所有源地址在 172.17.0.0/16 网段，目标地址为其他网段（外部网络）的流量动态伪装为从系统网卡发出。MASQUERADE 与传统 SNAT 的好处是它能动态地从网卡获取地址。

② 外部访问容器实现。

容器允许外部访问，可以在 docker run 的时候通过“-p”或“-P”参数来启用。

不管用何种方法，其实都是在本地的 iptables 的 NAT 表中添加相应的规则。

使用“-P”时：

```
# iptables -t nat -nL
...
Chain DOCKER （2 references）
target     prot opt source          destination
DNAT       tcp -- 0.0.0.0/0         0.0.0.0/0 tcp dpt:49153 to:172.17.0.2:80
```

使用“-p 80:80”时：

```
# iptables -t nat -nL
Chain DOCKER （2 references）
target     prot opt source          destination
DNAT       tcp  --  0.0.0.0/0       0.0.0.0/0 tcp dpt:80 to:172.17.0.2:80
```

注意

这里的规则映射了 0.0.0.0，意味着将接收主机所有接口的流量。用户可以通过-p IP:host_port:container_port 或-p IP::port 来指定允许访问容器的主机上的 IP、接口等，以制定更严格的规则。

在配置文件/etc/default/docker 中指定 DOCKER_OPTS="--ip=IP_ADDRESS"，之后重启 Docker 服务即可生效。

5）配置 docker0 网桥

Docker 服务默认会创建一个 docker0 网桥（其上有一个 docker0 内部接口），它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和本地主机都放到同一个物理网络。

Docker 默认指定了 docker0 接口的 IP 地址和子网掩码，让主机和容器之间可以通过网桥相互通信，它还给出了 MTU（接口允许接收的最大传输单元），通常是 1 500 Bytes，

版权所有，严禁传播，违者自负法律责任！

或宿主机网络路由上支持的默认值。这些值都可以在服务启动的时候进行配置。

--bip=CIDR -- IP 地址加掩码格式，如 192.168.1.5/24。

--mtu=BYTES --覆盖默认的 Docker mtu 配置。

用户也可以在配置文件中配置 DOCKER_OPTS，然后重启服务。由于目前的 Docker 网桥是 Linux 网桥，因此用户可以使用 brctl show 来查看网桥和端口连接信息。

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.3a1d7362b4ee       no              veth65f9 vethdda6
```

注意

brctl 命令在 Debian、Ubuntu 中可以使用 sudo apt-get install bridge-utils 来安装。

每当创建一个新容器的时候，Docker 就从可用的地址段中选择一个空闲的 IP 地址分配给容器的 eth0 端口。使用本地主机上 docker0 接口的 IP 作为所有容器的默认网关。

```
# docker run -i -t --rm base /bin/bash
# ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
       valid_lft forever preferred_lft forever
# ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3
# exit
```

6）自定义网桥

除了默认的 docker0 网桥，用户也可以指定网桥来连接各个容器。

在启动 Docker 服务的时候，可使用-b BRIDGE 或--bridge=BRIDGE 来指定使用的网桥。

如果服务已经运行，那么需要先停止服务，并删除旧的网桥。

```
# service docker stop
# ip link set dev docker0 down
# brctl delbr docker0
```

然后创建一个网桥 bridge0。

```
# brctl addbr bridge0
# ip addr add 192.168.5.1/24 dev bridge0
# ip link set dev bridge0 up
```

查看并确认网桥创建，然后启动。

```
$ ip addr show bridge0
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
```

版权所有，严禁传播，违者自负法律责任！


```
    inet 192.168.5.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
```

配置 Docker 服务，默认桥接到创建的网桥上。

```
# echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
# sudo service docker start
```

启动 Docker 服务，新建一个容器，可以看到它已经桥接到了 bridge0 上。

用户可以继续用 brctl show 命令查看桥接的信息。另外，在容器中可以使用 ip addr 和 ip route 命令来查看 IP 地址配置和路由信息。

7）介绍部分工具和示例

在介绍自定义网络拓扑之前，读者可能会对一些外部工具和例子感兴趣。

① pipework。

Jérôme Petazzoni 编写了一个名称为 pipework 的 shell 脚本，可以帮助用户在比较复杂的场景中完成容器的连接。

② playground。

Brandon Rhodes 创建了一个提供完整的 Docker 容器网络拓扑管理的 Python 库，包括路由、NAT 防火墙，以及一些提供 HTTP、SMTP、POP、IMAP、Telnet、SSH 和 FTP 的服务器。

8）编辑网络配置文件

Docker 1.2.0 开始支持在运行中的容器里编辑 /etc/hosts、/etc/hostname 和 /etc/resolve.conf 文件。

但是这些修改是临时的，只在运行的容器中保留，容器终止或重启后并不会被保存下来，也不会被 docker commit 提交。

9）创建一个点到点连接

默认情况下，Docker 会将所有容器连接到由 docker0 提供的虚拟子网中。

用户有时候需要两个容器之间可以直连通信，而不用通过主机网桥进行桥接。

解决办法很简单：创建一对 peer 接口，分别放到两个容器中，配置成点到点链路类型即可。

首先启动两个容器：

```
# docker run -i -t --rm --net=none base /bin/bash
root@1f1f4c1f931a:/#
# docker run -i -t --rm --net=none base /bin/bash
root@12e343489d2f:/#
```

找到进程号，然后创建网络命名空间的跟踪文件。

```
$ sudo docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a
2989
$ sudo docker inspect -f '{{.State.Pid}}' 12e343489d2f
3004
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/2989/ns/net /var/run/netns/2989
$ sudo ln -s /proc/3004/ns/net /var/run/netns/3004
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

创建一对 peer 接口，然后配置路由。

```
$ sudo ip link add A type veth peer name B

$ sudo ip link set A netns 2989
$ sudo ip netns exec 2989 ip addr add 10.1.1.1/32 dev A
$ sudo ip netns exec 2989 ip link set A up
$ sudo ip netns exec 2989 ip route add 10.1.1.2/32 dev A

$ sudo ip link set B netns 3004
$ sudo ip netns exec 3004 ip addr add 10.1.1.2/32 dev B
$ sudo ip netns exec 3004 ip link set B up
$ sudo ip netns exec 3004 ip route add 10.1.1.1/32 dev B
```

现在这两个容器就可以相互 ping 通，并成功建立连接。点到点链路不需要子网和子网掩码。

此外，也可以不指定--net=none 来创建点到点链路，这样，容器还可以通过原先的网络来通信。

利用类似的方法，可以创建一个只跟主机通信的容器。但是一般情况下，更推荐使用--icc=false 来关闭容器之间的通信。

项目实训

【实训题目】

启动 Ubuntu 容器，创建独立的网络命名空间，并创建虚拟网络接口设备 A，为 A 创建一个映射端设备 B，将设备 A 接入到创建的网桥 xd_br 中，完成后启动设备 A，将 B 设备放入刚刚创建好的网络空间中，完成后查询宿主机的网桥信息和该容器的网络信息。

【实训目的】

掌握 Docker 网络的各种配置方法，熟练使用满足不同需求时的不同配置。

【实训内容】

1. 使用 ubuntu:14.04.3 镜像启动 ubuntu 容器。
2. 使用 docker ps -a 命令查看容器 ID。
3. 使用命令获取容器的 pid 值，并为它创建网络命名空间。
4. 创建一对 veth pair 接口 A 和 B，绑定 A 到网桥 docker0，并启用它。
5. 将 B 放到容器的网络命名空间，命名为 eth0，启动它并配置一个可用 IP（桥接网段）和默认网关。
6. 查询宿主机的网桥信息和该容器的网络信息。

任务 4.3　Docker API 的使用

任务描述

1. 了解什么是 Docker API，了解 API 的主要类型和 3 个 Docker API 的功能。
2. 了解 Docker Remote API 的使用方法和操作步骤。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

笔 记

知识学习

1. Docker API 介绍

API 的主要类型包括 RESTful、基于 HTTP 协议和基于 RPC 协议。

所谓 RESTful，是指通过如 HTTP 协议封装的各种请求，主要包括 GET、POST、UPDATE、DELETE、PUT 等，保存创建、修改等操作。一般，RESTful 服务器提供这些请求的接口（地址、路径、参数），然后客户端可以通过 Linux 命令 curl，Python 标准库 httplib、httplib2 等访问。需要注意的是，多数参数（数据）都以 JSON/XML 打包，并且多数按照 token 方式验证请求权限和安全性。当然也支持其他类型的验证。

Docker 生态系统中一共有 3 种 API。

① Registry：提供了存储 Docker 镜像和集成 Docker Registry 的功能。

② Docker Hub API：提供了与 Docker Hub 集成的功能。

③ Docker Remote API：提供了与 Docker 守护进程进行集成的功能。

这 3 种 API 都是 RESTful 风格的。这里将会着重对 Remote API 进行介绍，因为它是通过程序与 Docker 进行集成和交互的核心内容。

2. 初识 remote API

首先浏览一下 Docker Remote API，并看看它都提供了哪些功能。要牢记的是，Remote API 是由 Docker 守护进程提供的。在默认情况下，Docker 守护进程会绑定到一个所在宿主机的套接字，即 unix:///var/run/docker.sock. docker 守护进程需要以 root 权限来运行，以便它有足够的权限去管理所需要的资源。也正如在之前所阐述的那样，如果系统中存在一个名为 docker 的用户组，那么 Docker 会将上面所说的套接字的所有者设为该用户组。因此任何属于 docker 用户组的用户都可以运行 Docker，而无须 root 权限。

如果只查询在同一台宿主机上运行 Docker 的 Remote API，那么上面的机制看起来没什么问题，但是如果想远程访问 Remote API，就需要将 Docker 守护进程绑定到一个网络接口上去。此时只需要给 Docker 守护进程传递一个-H 标志即可做到这一点。如果用户可以在本地使用 Docker API，那么就可以使用 nc 命令来进行查询。

在大多数操作系统上，用户可以通过编辑守护进程的启动配置文件将 Docker 守护进程绑定到指定网络接口。用户需要编辑/usr/lib/system/system/docker.service 文件。

任务实施

（1）remote API 的基本操作

现在来看看如何在一个运行 systemd 的 red hat 衍生版上将 Docker 守护进程绑定到一个网络接口上。编辑/usr/lib/system/system/docker.service 文件，将代码清单所示的内容修改为以下内容。

```
ExecStart=/usr/bin/docker-current -d --selinu-enabled -H tcp://0.0.0.0:2375
```

这将把 Docker 守护进程绑定到该宿主机的所有网络接口的 2375 端口上。之后需要使用 systemctl 命令来重新加载并启动该守护进程，如以下代码所示：

```
# systemctl --system daemon-reload
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

现在可以通过 Docker 客户端命令的-H 标志来测试一下刚才的配置是否生效。让一台远程主机来访问 Docker 守护进程，如以下代码所示。

```
# docker daemon -H unix:///var/run/docker.sock -H 0.0.0.0:2375
WARN[0000] /!\ DON'T BIND ON ANY IP ADDRESS WITHOUT setting -tlsverify IF YOU
DON'T KNOW WHAT YOU'RE DOING /!\
WARN[0000] devmapper: Usage of loopback devices is strongly discouraged for production
use. Please use '--storage-opt dm.thinpooldev' or use 'man docker' to refer to dm.thinpooldev section.
WARN[0000] devmapper: Base device already exists and has filesystem xfs on it. User
specified filesystem  will be ignored.
INFO[0000] [graphdriver] using prior storage driver "devicemapper"
INFO[0000] Graph migration to content-addressability took 0.00 seconds
INFO[0000] Firewalld running: false
INFO[0000] Default bridge （docker0）is assigned with an IP address 172.17.0.0/16. Daemon
option --bip can be used to set a preferred IP address
INFO[0000] Loading containers: start.
.
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon          commit=3999ccb-unsupported execdriver=native-0.2
graphdriver= devicemapper version=1.10.3
INFO[0001] API listen on [::]:2375
INFO[0001] API listen on /var/run/docker.sock


# docker -H 127.0.0.1:2375 info
Containers: 1
 Running: 1
 Paused: 0
 Stopped: 0
Images: 22
Server Version: 1.10.3
Storage Driver: devicemapper
 Pool Name: docker-253:1-285241608-pool
 Pool Blocksize: 65.54 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: xfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 5.105 GB
 Data Space Total: 107.4 GB
 Data Space Available: 90.75 GB
 Metadata Space Used: 10.23 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.137 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: false
 Deferred Deletion Enabled: false
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
 Deferred Deleted Device Count: 0
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 WARNING: Usage of loopback devices is strongly discouraged for production use. Either use
'--storage-opt dm.thinpooldev' or use '--storage-opt dm.no_warn_on_loop_devices=true' to suppress
this warning.
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
 Library Version: 1.02.135-RHEL7 (2016-09-28)
Execution Driver: native-0.2
Logging Driver: json-file
Plugins:
 Volume: local
 Network: host bridge null
Kernel Version: 3.10.0-229.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
Number of Docker Hooks: 2
CPUs: 1
Total Memory: 1.954 GiB
Name: docker
ID: HRBL:3YXE:HAAO:MRAG:NKQO:MZWC:7EVD:Q2QK:UYK6:C7NF:GOYU:ELSR
WARNING: bridge-nf-call-iptables is disabled
WARNING: bridge-nf-call-ip6tables is disabled
Registries: docker.io (secure)
```

这里假定 Docker 提供了更便捷的 docker_host 环境变量，这样就省掉了每次都需要设置-H 标志的麻烦。

```
# export DOCKER_HOST="tcp://127.0.0.1:2375"
```

（2）测试 Docker Remote API

现在已经通过 Docker 程序建立并确认了与 Docker 守护进程之间的网络连通性，接下来直接连接到 API。用 curl 命令查询得到的信息和用 docker info 命令得到的信息大致相同。

```
[root@docker ~]# curl http://127.0.0.1:2375/info
{"ID":"HRBL:3YXE:HAAO:MRAG:NKQO:MZWC:7EVD:Q2QK:UYK6:C7NF:GOYU:ELSR","Containers":1,"ContainersRunning":1,"ContainersPaused":0,"ContainersStopped":0,"Images":22,"Driver":"devicemapper","DriverStatus":[["Pool Name","docker-253:1-285241608-pool"],["Pool Blocksize","65.54 kB"],["Base Device Size","10.74 GB"],["Backing Filesystem","xfs"],["Data file","/dev/loop0"],["Metadata file","/dev/loop1"],["Data Space Used","5.106 GB"],["Data Space Total","107.4 GB"],["Data Space Available","90.75 GB"],["Metadata Space Used","10.23 MB"],["Metadata Space Total","2.147 GB"],["Metadata Space Available","2.137 GB"],["Udev Sync Supported","true"],["Deferred Removal Enabled", "false"],["Deferred Deletion Enabled","false"],["Deferred Deleted Device Count","0"],["Data loop file","/var/lib/docker/devicemapper/devicemapper/data"],["Metadata loop file","/var/lib/docker/devicemapper/ devicemapper/metadata"],["Library Version", "1.02.135-RHEL7 (2016-09-28)"]],"SystemStatus":null," Plugins":{"Volume":["local"],"Network":["host","bridge","null"],"Authorization":null},"MemoryLimit":true,"SwapLimit":true,"CpuCfsPeriod":true,"CpuCfsQuota":true,"CPUShares":true,"CPUSet":true,"IPv4Forwarding":true,"BridgeNfIptables":false,"BridgeNfIp6tables":false,"Debug":false,"NFd":21,"OomKillDisable":true,"NGorou tines":28,"System
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

Time":"2016-12-30T03:24:27.654171066Z","ExecutionDriver":"native-0.2","LoggingDriver":"json-file","NEventsListener":0,"KernelVersion":"3.10.0-229.el7.x86_64","PkgVersion":"docker-common-1.10.3 -59.el7.CentOS.x86_64","OperatingSystem":"CentOS Linux 7 (Core)","OSType":"linux","Architecture":"x86_64","IndexServerAddress":"https://index.docker.io/v1/","IndexServerName":"docker.io", "RegistryConfig":{"InsecureRegistryCIDRs":["127.0.0.0/8"],"IndexConfigs":{"docker.io": {"Name":"docker.io","Mirrors":null,"Se- cure":true,"Official":true}},"Mirrors":null},"InitSha1":"2195bfa7012bb8ffc50f899b658b487b565336e1","InitPath":"","NCPU":1,"MemTotal":2098581504,"DockerRootDir":"/var/lib/docker","HttpProxy":"","HttpsProxy":"","NoProxy":"","Name":"docker","Labels":null,"ExperimentalBuild":false,"ServerVersion":"1.10.3","ClusterStore":"","ClusterAdvertise":"","Registries":[{"Name":"docker.io","Secure":true}]}

① 通过 API 来管理 Docker 镜像。

操作 Docker 镜像的 API，将获取 Docker 守护进程中所有镜像的列表：

```
# curl http://127.0.0.1:2375/images/json | python -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  6781    0  6781    0     0   245k      0 --:--:-- --:--:-- --:--:--  254k
[
    {
        "Created": 1482998541,
        "Id": "sha256:0c27604c726690da5f7368af31765903bbf196b8e5808c876a2abd25b681a17f",
        "Labels": null,
        "ParentId": "",
        "RepoDigests": null,
        "RepoTags": [
            "odoo:latest"
        ],
        "Size": 0,
        "VirtualSize": 0
    },
    {
        "Created": 1482966589,
        "Id": "sha256:411afb6272fbf5258c421ca5dd10bf212cddb40d78ac3962114b67bb78cc7c62",
        "Labels": {},
        "ParentId": "",
        "RepoDigests": [
            "docker.io/python@sha256:05d7f756d1980a83ca7419109334a6df47824e02eae93996c3d201f96baf7cf3"
        ],
        "RepoTags": [
            "docker.io/python:latest"
        ],
        "Size": 687122335,
        "VirtualSize": 687122335
},
```

这里使用了/images/json 这个接入点，它将返回 Docker 守护进程中的所有镜像的列表。

最后，也像命令行一样，也可以在 Docker Hub 上查找镜像。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
# curl http://192.168.200.22:2375/images/search?term="jamtur01" | python -mjson.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  4387    0  4387    0     0   1023      0 --:--:--  0:00:04 --:--:--  1023
[
    {
        "description": "",
        "index_name": "docker.io",
        "is_automated": true,
        "is_official": false,
        "is_trusted": true,
        "name": "jamtur01/docker-presentation",
        "registry_name": "docker.io",
        "star_count": 5
    },
    {
        "description": "",
        "index_name": "docker.io",
        "is_automated": false,
        "is_official": false,
        "is_trusted": false,
        "name": "jamtur01/dockerjenkins",
        "registry_name": "docker.io",
        "star_count": 3
    },
```

这只是使用 Docker API 能完成的工作的一个例子而已，实际上还能用 API 进行镜像构建、更新和删除。

② 通过 API 管理 Docker 容器。

Docker Remote API 同样也可以使用/container 参数列出所有的容器。

```
# curl -s "http://127.0.0.1:2375/containers/json" | python -m json.tool
[
    {
        "Command": "/entrypoint.sh /etc/docker/registry/config.yml",
        "Created": 1483003943,
        "HostConfig": {
            "NetworkMode": "default"
        },
        "Id": "09dda7fe410ba278db18ca1c0aade605083ac6151ce6e005527304f7cc3c5331",
        "Image": "docker.io/registry:latest",
        "ImageID": "sha256:c9bd19d022f6613fa0e3d73b2fe2b2cffe19ed629a426db9a652b59
7fccf07d4",
        "Labels": {},
        "Names": [
            "/registry"
        ],
        "NetworkSettings": {
            "Networks": {
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

```
                    "bridge": {
                        "Aliases": null,
                        "EndpointID":
"98594476f7c321b1301ea67aadbadbe412a6d44e49e3d6fb5ee152ba12eaab7f",
                        "Gateway": "172.17.0.1",
                        "GlobalIPv6Address": "",
                        "GlobalIPv6PrefixLen": 0,
                        "IPAMConfig": null,
                        "IPAddress": "172.17.0.2",
                        "IPPrefixLen": 16,
                        "IPv6Gateway": "",
                        "Links": null,
                        "MacAddress": "02:42:ac:11:00:02",
                        "NetworkID": ""
                    }
                }
            },
            "Ports": [
                {
                    "IP": "0.0.0.0",
                    "PrivatePort": 5000,
                    "PublicPort": 5000,
                    "Type": "tcp"
                }
            ],
            "Status": "Up 53 minutes"
        }
    ]
```

也可以通过使用 POST 请求来调用/containers/create 接入点来创建容器。

```
    # curl -X POST -H "Content-Type: application/json"    http://127.0.0.1:2375/containers/create -d
'{"image":"docker.io/python"}'
    {"Id":"e7f1abb491ebfb1bb71b81ec9cb0eb5f72e7d4ce03bcaa5b491565c663ff1929","Warnings":null}
```

这里调用了/containers/create 接入点，并 POST 了一个 json 散列数据，这个结构中包括要启动的镜像名。这个 API 返回了刚创建的容器的 ID，以及可能的警告信息。这条命令将会创建一个容器。

项目实训

【实训题目】

通过 API 来查找 Docker 镜像列表和创建容器。

【实训目的】

掌握 Docker API 的使用方法，可以通过 API 来管理 Docker 镜像和容器。

非卖品，仅供非商业用途或交流学习使用

笔 记

【实训内容】

1. 编辑 Docker 配置文件/usr/lib/system/system/docker.service，将 Docker 守护进程绑定到一个网络接口上。
2. 之后需要使用 systemctl 命令来重新加载并启动该守护进程。
3. 确认通过 Docker 程序建立并确认了与 Docker 守护进程之间的网络连通性。
4. 接下来直接连接到 API，然后通过 API 来查找 Docker 镜像信息。
5. 使用/container 接入点列出所有容器。
6. 使用 POST 请求来调用/containers/create 接入点来创建容器。

任务 4.4　Docker 底层技术应用

任务描述

1. 了解什么是命名空间，并会使用命名空间。
2. 熟悉控制组（Cgroups）的使用。
3. 了解联合文件系统（UnionFS）。
4. 熟悉容器的基本管理。

知识学习

1. 内核命名空间

Docker 容器和 LXC 容器很相似，所提供的安全特性也差不多。当用 docker run 启动一个容器时，在后台，Docker 为容器创建了一个独立的命名空间和控制组集合。

命名空间提供了最基础和最直接的隔离，在容器中运行的进程不会被运行在主机上的进程和其他容器发现和作用。

每个容器都有自己独有的网络栈，意味着它们不能访问其他容器的 sockets 或接口。不过，如果主机系统上做了相应的设置，则容器可以像跟主机交互一样地和其他容器交互。当指定公共端口或使用 links 来连接两个容器时，容器就可以相互通信了（可以根据配置来限制通信的策略）。

从网络架构的角度来看，所有的容器通过本地主机的网桥接口相互通信，就像物理机器通过物理交换机通信一样。

那么，内核中实现命名空间和私有网络的代码是否足够成熟?

内核命名空间从 2.6.15 版本（2008 年 7 月发布）开始被引入，数年间，这些机制的可靠性在诸多大型生产系统中被实践验证。

实际上，命名空间的想法和设计提出的时间更早，最初是为了在内核中引入一种机制来实现OpenVZ的特性。而 OpenVZ 项目早在 2005 年就发布了，目前其设计和实现都已经十分成熟。

2. 控制组（Cgroups）简介

控制组是 Linux 容器机制的另外一个关键组件，负责实现资源的审计和限制。

非卖品，仅供非商业用途或交流学习使用

笔记

它提供了很多有用的特性，可以确保各个容器可以公平地分享主机的内存、CPU、磁盘 IO 等资源。当然，更重要的是，当容器内的资源使用产生压力时，控制组确保了不会连累主机系统。

尽管控制组不负责隔离容器之间相互访问、处理数据和进程，它在防止拒绝服务（DDOS）攻击方面却是必不可少的。尤其是在多用户的平台（比如公有或私有的 PaaS）上，控制组十分重要。例如，当某些应用程序表现异常的时候，可以保证一致地正常运行。

控制组机制始于 2006 年，内核从 2.6.24 版本开始被引入。

Cgroups 是 Linux 内核提供的一种机制，它可以根据特定的行为把一系列系统任务及其子任务整合或分隔到不同的资源等级分组中，从而为系统资源管理提供一个统一的框架。

（1）控制组的主要功能

① 资源限制（Resource Limitation）：Cgroups 可以对进程组使用的资源总额进行限制。如设定应用运行时使用内存的上限，一旦超过这个配额就发出 OOM（Out of Memory）。

② 优先级分配（Prioritization）：可以分配 CPU 时间片数量及硬盘 IO 带宽大小，实际上就相当于控制了进程运行的优先级。

③ 资源统计（Accounting）：Cgroups 可以统计系统的资源使用量，如 CPU 使用时长、内存用量等。这个功能非常适用于计费。

④ 进程控制（Control）：Cgroups 可以对进程组执行挂起、恢复等操作。

通俗地来说，Cgroups 可以限制、记录、隔离进程组所使用的物理资源（包括 CPU、memory、IO 等），为容器实现虚拟化提供了基本保证，是构建 Docker 等一系列虚拟化管理工具的基础。

（2）Cgroups 功能的实现特点

① Cgroups 的 API 以一个伪文件系统的方式实现，即用户可以通过文件操作实现 Cgroups 的组织管理。

② Cgroups 的组织管理操作单元可以达到线程级别，用户态代码也可以针对系统分配的资源创建和销毁 Cgroups，从而实现资源再分配和管理。

③ 所有资源管理的功能都以 subsystem（子系统）的方式实现，接口统一。

④ 子进程创建之初与其父进程处于同一个控制组。

Cgroups 的实现本质上是给系统进程挂上钩子（hooks），当 task 运行的过程中涉及某个资源时就会触发钩子上所附带的 subsystem 进行检测，最终根据资源类别的不同使用对应的技术进行资源限制和优先级分配。

（3）Cgroups 文件讲解及基本使用

Cgroups：Control Groups，用于限制和隔离一组进程对系统资源的使用。它能实现资源的 QoS（Quality of Service），资源包括 CPU、内存、block I/O 和网络带宽。

Cgroups 中的子系统如下。

① device：设备权限控制。

② cpuset：分配指定的 CPU 和内存节点。

③ cpu：控制 CPU 占用率。

④ cpuacct：统计 CPU 的使用情况。

⑤ memory：限制内存的使用上限。

非卖品，仅供非商业用途或交流学习使用

笔记

⑥ freezer：冻结 Cgroups 中的进程。

⑦ net_cls：配合 TC（Traffic Controller）限制网络带宽。

⑧ net_prio：设置进程的网络流量优先级。

⑨ huge_tlb：限制 HugeTLB 的使用。

⑩ perf_event：允许 Perf 工具基于 Cgroups 分组做性能检测。

在 Cgroups 出现之前，只能对一个进程做资源控制（Cgroups 可以对任意组进程进行资源控制），例如用 sched_setaffinity 的系统调用来限定一个进程的 CPU 亲和性，或用 ulimit 来限制进程打开文件的上限、栈大小等。

（4）服务端的防护

运行一个容器或应用程序的核心是通过 Docker 服务端。Docker 服务的运行目前需要 root 权限，因此其安全性十分关键。

首先，确保只有可信的用户才可以访问 Docker 服务。Docker 允许用户在主机和容器间共享文件夹，同时不需要限制容器的访问权限，这就容易让容器突破资源限制。例如，恶意用户启动容器的时候将主机的根目录映射到容器的“/host”目录中，那么理论上容器就可以对主机的文件系统进行任意修改了。这听起来很疯狂，但是事实上几乎所有的虚拟化系统都允许类似的资源共享，而没法禁止用户共享主机根文件系统到虚拟机系统。

这将会造成很严重的安全后果。因此，当提供容器创建服务时（如通过一个 Web 服务器），要更加注意进行参数的安全检查，防止恶意的用户用特定参数来创建一些破坏性的容器。

为了加强对服务端的保护，Docker 的 REST API（客户端用来跟服务端通信）在 0.5.2 版本之后使用本地的 UNIX 套接字机制替代了原先绑定在 127.0.0.1 上的 TCP 套接字，因为后者容易遭受跨站脚本攻击。现在，用户使用 UNIX 权限检查来加强套接字的访问安全。

用户仍可以利用 HTTP 提供 REST API 访问。建议使用安全机制，确保只有可信的网络、VPN、证书保护机制（如受保护的 stunnel 和 ssl 认证）下的访问可以进行。此外，还可以使用 HTTPS 和证书来加强保护。

最近改进的 Linux 命名空间机制将可以实现非 root 用户来运行全功能的容器。这将从根本上解决了容器和主机之间共享文件系统而引起的安全问题。

终极目标是改进两个重要的安全特性：

① 将容器的 root 用户映射到本地主机上的非 root 用户，减轻容器和主机之间因权限提升而引起的安全问题；

② 允许 Docker 服务端在非 root 权限下运行，利用安全可靠的子进程来代理执行需要特权权限的操作。这些子进程将只允许在限定范围内进行操作，例如仅仅负责虚拟网络设定或文件系统管理、配置操作等。

最后，建议采用专用的服务器来运行 Docker 和相关的管理服务（如 ssh 监控和进程监控、管理工具 nrpe、collectd 等管理服务）。其他业务服务都放到容器中去运行。

（5）内核能力机制

能力机制（Capability）是 Linux 内核的一个强大特性，可以提供细粒度的权限访问控制。Linux 内核自 2.2 版本起就支持能力机制，它将权限划分为更加细粒度的操作能力，既可以作用在进程上，也可以作用在文件上。

例如，一个 Web 服务进程只需要绑定一个低于 1024 的端口的权限，并不需要 root

权限，那么它只需要被授权 net_bind_service 能力即可。此外，还有很多其他的类似能力来避免进程获取 root 权限。

笔记

默认情况下，Docker 启动的容器被严格限制为只允许使用内核的一部分能力。

使用能力机制对加强 Docker 容器的安全有很多好处。通常，在服务器上会运行一堆需要特权权限的进程，包括 ssh、cron、syslogd、硬件管理工具模块（如负载模块）、网络配置工具等。容器与这些进程是不同的，因为几乎所有的特权进程都由容器以外的支持系统来进行管理。

① ssh 访问被主机上的 ssh 服务来管理。

② cron 通常应该作为用户进程执行，权限交给使用它服务的应用来处理。

③ 日志系统可由 Docker 或第三方服务管理。

④ 若硬件管理无关紧要，则容器中也就无须执行 udevd 及类似服务。

⑤ 网络管理也都在主机上进行，除非特殊需求，容器不需要对网络进行配置。

从上面的例子可以看出，大部分情况下，容器并不需要“真正的”root 权限，容器只需要少数的能力即可。为了加强安全，容器可以禁用一些没必要的权限。

① 完全禁止任何 mount 操作。

② 禁止直接访问本地主机的套接字。

③ 禁止访问一些文件系统的操作，比如创建新的设备、修改文件属性等。

④ 禁止模块加载。

这样，就算攻击者在容器中取得了 root 权限，也不能获得本地主机的较高权限，能进行的破坏也有限了。

默认情况下，Docker 采用白名单机制，禁用必需功能之外的其他权限。当然，用户也可以根据自身需求来为 Docker 容器启用额外的权限。

（6）其他安全特性

除了能力机制之外，还可以利用一些现有的安全机制来增强使用 Docker 的安全性，如 TOMOYO、AppArmor、SELinux、GRSEC 等。

Docker 默认只启用了能力机制。用户可以采用多种方案来加强 Docker 主机的安全，例如在内核中启用 GRSEC 和 PAX，这将增加很多编译和运行时的安全检查；通过地址随机化来避免恶意探测等，并且启用该特性不需要 Docker 进行任何配置。

使用一些具有增强安全特性的容器模板，比如带 AppArmor 的模板、Redhat 系列带 SELinux 策略的模板。这些模板提供了额外的安全特性。用户可以自定义访问控制机制来定制安全策略。

跟其他添加到 Docker 容器的第三方工具一样，有很多类似的机制，在不改变 Docker 内核的情况下就可以加固现有的容器。

总体来看，Docker 容器还是十分安全的，特别是在容器内不使用 root 权限来运行进程的情况下。

另外，用户可以使用现有的工具，比如 AppArmor、SELinux 和 GRSEC，来增强安全性，甚至自己在内核中实现更复杂的安全机制。

3. 联合文件系统（UnionFS）

联合文件系统（UnionFS）是一种分层、轻量级且具有高性能的文件系统，它支持对文件系统修改时按层（Layer）叠加，同时可以将不同目录挂载到同一个虚拟文件系统下。

联合文件系统是 Docker 镜像的基础。镜像可以通过分层来进行继承。基于基础镜像，可以制作各种具体的应用镜像。Aufs 是 Docker 最初采用的文件系统，Aufs 未能加入到 Linux 内核，考虑到兼容性问题，加入了 DeviceMapper 的支持。Docker 目前支持的联合文件系统包括 Aufs、btrfs、vfs 和 DeviceMapper。

典型的 Linux 文件系统由 bootfs 和 rootfs 两部分组成。bootfs（Boot File System）主要包含 bootloader 和 kernel，bootloader 主要是引导加载 kernel，当 kernel 被加载到内存中后，bootfs 就被 umount 了。rootfs（Root File System）包含的就是典型 Linux 系统中的/dev、/proc、/bin、/etc 等标准目录和文件。图 4-4-1 所示为 Linux 加载结构。

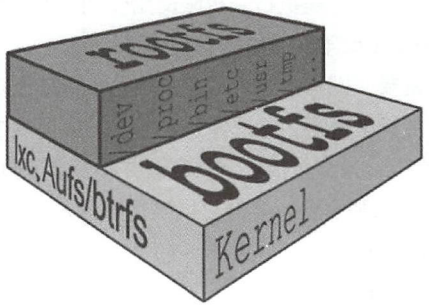


图 4-4-1
Linux 加载结构

Docker 容器建立在 Aufs 基础上，Aufs 是一种 UnionFS，简单来说就是支持将不同的目录挂载到同一个虚拟文件系统下，并实现一种 Layer 的概念。Aufs 将挂载到同一虚拟文件系统下的多个目录分别设置为 read-only、read-write 及 whiteout-able 权限，对 read-only 目录只能读，而写操作只能实施在 read-write 目录中。写操作是对 read-only 上的一种增量操作，不影响 read-only 目录。挂载目录时要严格按照各目录之间的这种增量关系，将被增量操作的目录优先于在它基础上进行增量操作的目录来挂载，待所有目录挂载结束了，继续挂载一个 read-write 目录，如此便形成了一种层次结构。

Docker 镜像的典型结构如图 4-4-2 所示。传统的 Linux 加载 bootfs 时会先将 rootfs 设置为 read-only，然后在系统自检之后将 rootfs 从 read-only 改为 read-write，然后就可以在 rootfs 上进行写和读的操作了。

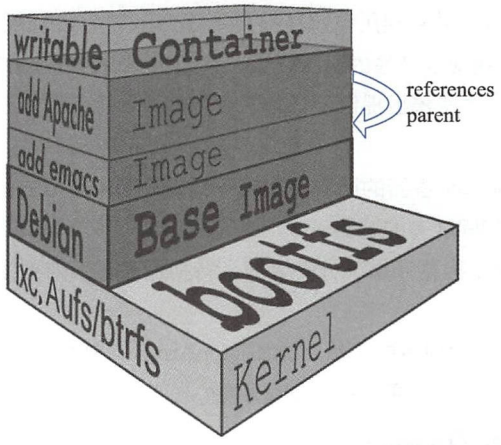


图 4-4-2
Docker 镜像结构

但 Docker 的镜像却不这样，在 bootfs 自检完毕之后，它并不会把 rootfs 的 read-only 改为 read-write，而是利用 union mount（UnionFS 的一种挂载机制）将一个或多个 read-only

的 rootfs 加载到之前的 read-only 的 rootfs 层之上。在加载了这么多层的 rootfs 之后，仍然让它看起来是一个文件系统，在 Docker 的体系里把 union mount 的这些 read-only 的 rootfs 称为 Docker 的镜像。但是，此时的每一层 rootfs 都是 read-only 的，还不能对其进行操作。当创建一个容器时，也就是将 Docker 镜像进行实例化时，系统会在一层或是多层 read-only 的 rootfs 之上分配一层空的 read-write 的 rootfs。

为了形象化 Docker 的镜像结构，这里 docker pull 一个 ubuntu:14.04 的镜像，使用 docker images -tree 查看的结果如下：

```
# docker images -tree
Warning: '-tree' is deprecated, it will be removed soon. See usage.
└─511136ea3c5a Virtual Size: 0 B
  └─3b363fd9d7da Virtual Size: 192.5 MB
    └─607c5d1cca71 Virtual Size: 192.7 MB
      └─f62feddc05dc Virtual Size: 192.7 MB
        └─8eaa4ff06b53 Virtual Size: 192.7 MB Tags: ubuntu:14.04,
```

在高 Docker 版本下，docker image 不再使用，可以使用“docker history 镜像 ID”来查看。

可以看到 Ubuntu 的镜像中有多个长 ID 的 Layer，且以一种树状结构继承下来，如图 4-4-3 所示。其中，第 n+1 层继承了第 n 层，并在此基础上有了自己的内容，直观上的表现就是第 n+1 层占用的磁盘空间增大。并且，不同的镜像可能会有相同的父镜像。Tomcat 和 Nginx 继承于同一个 Vim 镜像，这种组织方式起到共享的作用，减少了镜像在物理机上占用的空间。

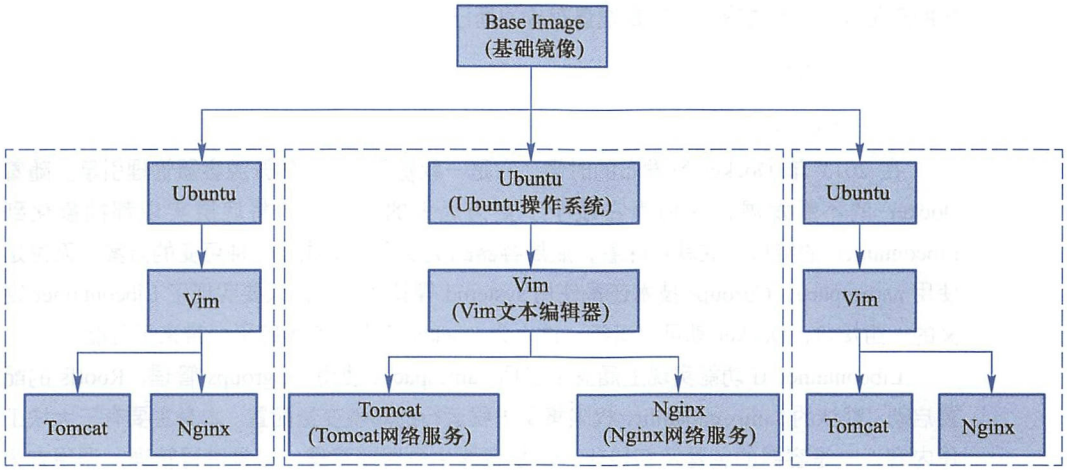


图 4-4-3 镜像继承关系

使用 docker save 命令保存镜像 tar，并查询其结构。

```
# docker save -o ubuntu.tar ubuntu:14.04
#   tar -tf ubuntu.tar
3b363fd9d7dab4db9591058a3f43e806f6fa6f7e2744b63b2df4b84eadb0685a/
3b363fd9d7dab4db9591058a3f43e806f6fa6f7e2744b63b2df4b84eadb0685a/VERSION
3b363fd9d7dab4db9591058a3f43e806f6fa6f7e2744b63b2df4b84eadb0685a/json
3b363fd9d7dab4db9591058a3f43e806f6fa6f7e2744b63b2df4b84eadb0685a/layer.tar
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158/
```

```
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158/VERSION
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158/json
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158/layer.tar
607c5d1cca71dd3b6c04327c3903363079b72ab3e5e4289d74fb00a9ac7ec2aa/
607c5d1cca71dd3b6c04327c3903363079b72ab3e5e4289d74fb00a9ac7ec2aa/VERSION
607c5d1cca71dd3b6c04327c3903363079b72ab3e5e4289d74fb00a9ac7ec2aa/json
607c5d1cca71dd3b6c04327c3903363079b72ab3e5e4289d74fb00a9ac7ec2aa/layer.tar
8eaa4ff06b53ff7730c4d7a7e21b4426a4b46dee064ca2d5d90d757dc7ea040a/
8eaa4ff06b53ff7730c4d7a7e21b4426a4b46dee064ca2d5d90d757dc7ea040a/VERSION
8eaa4ff06b53ff7730c4d7a7e21b4426a4b46dee064ca2d5d90d757dc7ea040a/json
8eaa4ff06b53ff7730c4d7a7e21b4426a4b46dee064ca2d5d90d757dc7ea040a/layer.tar
f62feddc05dc67da9b725361f97d7ae72a32e355ce1585f9a60d090289120f73/
f62feddc05dc67da9b725361f97d7ae72a32e355ce1585f9a60d090289120f73/VERSION
f62feddc05dc67da9b725361f97d7ae72a32e355ce1585f9a60d090289120f73/json
f62feddc05dc67da9b725361f97d7ae72a32e355ce1585f9a60d090289120f73/layer.tar
repositories
```

每一个 Layer 都有一个文件夹存在，且每个文件夹中都包含 VERSION、json 和 layer.tar 这 3 个文件，另外还有一个 repositories 文件。长 ID 为 511136ea3c5a 的 Layer 没有继承任何 Layer，因此被称为 base 镜像。一般来说，镜像都是以这个 Layer 开始的，其内容也为空。

json 文件包含了镜像及其对应的容器的配置信息，其中进行 docker run 操作时指定的，如 CpuShares 和 Cpuset，也有系统生成的如 parent，正是这个属性记录了继承结构，从而实现了 Layer 的层次关系。每个文件夹下都有一个 layer.tar 的文件，这个压缩文件存放的正是 rootfs 的内容，不过是增量存放的而已。

4. 容器管理

（1）Libcontainer

在 2013 年 Docker 刚发布的时候，它是一款基于 LXC 的开源容器管理引擎。随着 Docker 的不断发展，人们为它赋予了更为远大的目标，即将底层实现都抽象化到 Libcontainer 的接口。这就意味着，底层容器的实现方式变成了一种可变的方案，无论是使用 namespace、Cgroups 技术还是使用 systemd 等其他方案，只要实现了 Libcontainer 定义的一组接口，Docker 都可以运行。这也为 Docker 实现全面地跨平台带来了可能。

Libcontainer 在功能实现上涵盖了包括 namespaces 使用、Cgroups 管理、Rootfs 的配置启动、默认的 Linux capability 权限集及进程运行的环境变量配置。本身主要有三大块工作内容，一是容器的创建及初始化，二是容器生命周期管理，三是进程管理，调用方为 Docker 的 execdriver。容器的监控主要通过 Cgroups 的状态统计信息来实现，未来会加入进程追踪等更丰富的功能。另一方面，Libcontainer 在安全支持方面也为用户尽可能多地提供了支持和选择。

（2）runC

Linux 基金会于 2015 年 6 月成立 OCI（Open Container Initiative）组织，旨在围绕容器格式和运行时制定一个开放的工业化标准。OCI 的目标是为了避免容器的生态分裂为“小生态王国”，确保一个引擎上构建的容器可以运行在其他引擎之上。这是实现容器可移植性至关重要的部分。该组织一成立便得到了包括谷歌、微软、亚马逊、华为等一系列云

计算厂商的支持。

OCI 推进容器标准化具体完成的内容如下。

① 操作标准化：容器的标准化操作包括使用标准容器创建、启动、停止容器，使用标准文件系统工具复制和创建容器快照，使用标准化网络工具进行下载和上传。

② 内容无关：内容无关指不管针对的具体容器内容是什么，容器标准操作执行后都能产生同样的效果。如容器可以用同样的方式上传、启动，不管是 PHP 应用还是 MySQL 数据库服务。

③ 基础设施无关：无论是个人的便携式计算机还是 AWS S3，抑或是 Openstack，或者其他基础设施，都应该支持容器的各项操作。

④ 为自动化量身定制：制定容器统一标准，是操作内容无关化、平台无关化的根本目的之一，就是为了可以使容器操作全平台自动化。

⑤ 工业级交付：是制定容器标准的一大目标，就是使软件分发达到工业级交付成为现实。

runC 是 Docker 贡献的一个 CLI 工具，通过对 Libcontainer 的封装，采用了开放容器格式标准（Open Container Format，OCF）。2016 年 6 月份发布 runC1.0 版本。

（3）Containerd

2016 年 12 月，Docker 将 Docker Engine 中的核心元件 Containerd 开源，作为一个独立的开源项目独立发展，目标是提供一个更加开放、稳定的容器运行基础设施。Containerd 可以说是 Docker 引擎的引擎（Runtime）。和包含在 Docker Engine 里 Containerd 相比，独立的 Containerd 将具有更多的功能，可以满足整个容器运行时管理的所有需求。图 4-4-4 所示为镜像、容器、runC、Containerd 自己的调用关系。

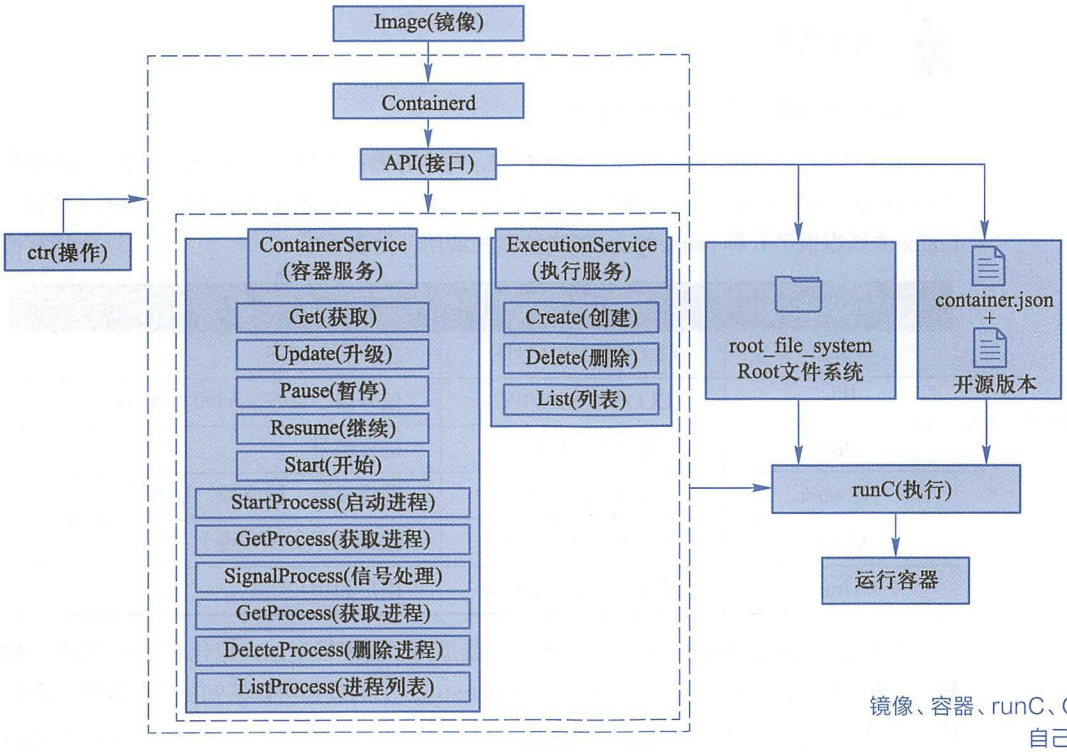


图 4-4-4
镜像、容器、runC、Containerd
自己的调用关系

Containerd 面向运维人员而构建，目的是构建工业级规模的 Docker 系统，在性能上进行了优化，Containerd 利用 runC 来提供高级功能，如热迁移（Checkpoint & Restore）seccomp 及用户命名空间支持，这将为 Docker 引入这些功能敞开大门。Containerd 的架构如图 4-4-5 所示。

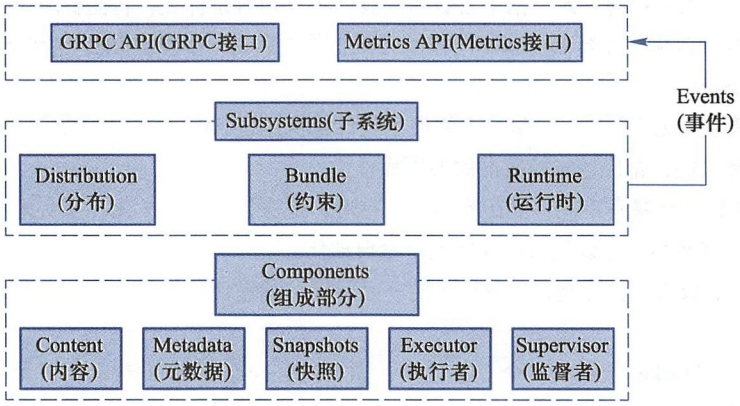


图 4-4-5
Containerd 架构图

Containerd 分 3 层，最上层为 API，包括 GRPC API（Go RPC）和 Metrics API。

中间层为内部子系统，包括以下 3 个部分。

① Distribution：和 Docker Registry 打交道，拉取镜像。

② Bundle：管理本地磁盘上镜像的子系统。

③ Runtime：创建容器，管理容器的子系统。

最下层为功能组件，包括内容、元数据、快照、执行器和监控器。

任务实施

1. 使用命名空间（NameSpace）

命名空间是 Linux 内核的一个强大特性。每个容器都有自己单独的命名空间，运行在其中的应用都像在独立的操作系统中运行一样。命名空间保证了容器之间彼此互不影响。Linux 内核提供了 6 种 NameSpace 隔离的系统调用，见表 4-1。

表 4-1　NameSpace 的 6 项隔离

NameSpace	系统调用参数	隔 离 内 容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

Linux 内核实现 NameSpace 的主要目的就是为了实现轻量级虚拟化（容器）服务，构建一个相对隔离的 shell 环境。在同一个 NameSpace 下的进程可以感知彼此的变化，而对外界的进程则一无所知，这样可让容器内的进程运行在一个独立的系统环境中，以此达到隔离的目的。

Namespace 的基本操作如下。

① 查看容器的进程 Pid。

```
#docker inspect --format '{{.State.Pid}}' web5
```

② 建立命名空间 web5 与容器之间的连接。

```
# mkdir /var/run/netns
# ln -s /proc/2704/ns/net /var/run/netns/web5
```

③ 查看命名空间。

```
#ip netns
Web5
```

④ 查看命名空间 IP，即容器 IP。

```
ip netns exec web5 ip addr
```

⑤ 添加命名空间 IP。

```
ip netns exec web5   ip addr add local 172.17.0.4/24 brd + label eth0:1 dev eth0
```

2. 使用和管理 Cgroups

(1) Cgroups 的接口与使用

现在可以看一下 Cgroupsfs 虚拟文件系统中的内容，本书的环境是 CentOS7。

```
# cd /sys/fs/Cgroups/
# ls
blkio cpu cpuacct cpu,cpuacct cpuset devices freezer hugetlb memory net_cls perf_event systemd
```

Cgroups 目录下的文件就是上述所说的子系统，就是由这些子系统对资源进行着限制。

```
cpuset
# cd cpuset/
# ls
Cgroups.clone_children cpuset.cpu_exclusive cpuset.memory_migrate cpuset.memory_spread_slabDocker
Cgroups.event_control cpuset.cpus cpuset.memory_pressure cpuset.mems notify_on_release
Cgroups.procs cpuset.mem_exclusive cpuset.memory_pressure_enabled cpuset.sched_load_balance
release_agent
Cgroups.sane_behavior cpuset.mem_hardwall cpuset.memory_spread_page cpuset.sched_relax_domain_ level tasks
```

此时能够看到里面有非常多的控制文件。

```
# cat cpuset.cpus
0
# cat cpuset.mems
0
```

在这里可以看到一些文件中的内容，这两个控制文件的内容表达的含义是，限制这个 Cgroups 的进程只能在 0 号 CPU 上运行，且只会从 0 号内存节点分配内存。

非卖品，仅供非商业用途或交流学习使用

这里再关注一个文件 tasks，里面记录这个 Cgroups 的所有进程（包括线程）。

```
# cat tasks
1
2
3
…
7
8
9
```

如何创建一个 Cgroups，很简单，在 Cgroups 目录下的子系统中创建一个目录“mkdir /sys/fs/Cgroups/cpuset/child/”，这样就创建了一个新的 Cgroups，开发者通过查看目录内容可以发现目录下自动创建了控制文件。

```
ls /sys/fs/Cgroups/cpuset/child/
Cgroups.clone_children cpuset.cpu_exclusive cpuset.mem_hardwall cpuset.memory_
spread_page cpuset. sched_load_balance tasks
Cgroups.event_control cpuset.cpus cpuset.memory_migrate cpuset.memory_spread_slab cpuset.
sched_ relax_domain_level
Cgroups.procs cpuset.mem_exclusive cpuset.memory_pressure cpuset.mems notify_on_release
```

开发者可以自己设置控制文件中的内容：

```
echo 0 > /sys/fs/Cgroups/cpuset/child/cpuset.cpus
echo 0 > /sys/fs/Cgroups/cpuset/child/cpuset.mems
echo $$ > /sys/fs/Cgroups/cpuset/child/tasks
```

把根目录当前进程 ID 写入 tasks 文件，使当前进程移动到这个 Cgroups 中，并且以后从这个进程派生的子进程也会被移入这个 Cgroups 中。

注意

开发者也可以将 pid 写入 Cgroups.procs，区别是 tasks 只会把指定的进程放入 child 中，Cgroups.procs 则会把这个进程所属的整个线程组都加到 child 中。

```
# echo $$ > tasks
# cat tasks
11417
30522
# ps
PID TTY TIME CMD
11424 pts/1 00:00:00 ps
30522 pts/1 00:00:00 bash
# ps -aux | grep 30522
root 11556 0.0 0.0 112664 976 pts/1 R+ 17:32 0:00 grep –color=auto 30522
root 30522 0.0 0.3 116836 3680 pts/1 Ss 15:47 0:00 –bash
```

（2）CPU 资源配额控制之 CPU 份额控制

Docker 提供了–cpu-shares 参数，可在创建容器时指定容器所使用的 CPU 份额值。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

使用示例：

使用命令 docker run -tid –cpu-shares 100 ubuntu:stress 创建容器，则最终生成的 Cgroups 的 CPU 份额配置可以在下面的文件中找到：

```
root@ubuntu:~# cat /sys/fs/Cgroups/cpu/docker/<容器的完整长 ID>/cpu.shares
```

–cpu-shares 的值不能保证可以获得一个 vcpu 或者多少 GHz 的 CPU 资源，仅仅只是一个弹性的加权值。

默认情况下，每个 Docker 容器的 CPU 份额都是 1 024。单独一个容器的份额是没有意义的，只有在同时运行多个容器时，容器的 CPU 加权的效果才能体现出来。例如，两个容器 A、B 的 CPU 份额分别为 1 000 和 500，在 CPU 进行时间片分配的时候，容器 A 比容器 B 多一倍的机会获得 CPU 的时间片，但分配的结果取决于当时主机和其他容器的运行状态，实际上也无法保证容器 A 一定能获得 CPU 时间片。比如容器 A 的进程一直是空闲的，那么容器 B 是可以获取比容器 A 更多的 CPU 时间片的。极端情况下，比如说，主机上只运行了一个容器，即使它的 CPU 份额只有 50，它也可以独占整个主机的 CPU 资源。

Cgroups 只在容器分配的资源紧缺时，也就是说在需要对容器使用的资源进行限制时，才会生效。因此，无法单纯根据某个容器的 CPU 份额来确定有多少 CPU 资源分配给它，资源分配结果取决于同时运行的其他容器的 CPU 分配和容器中进程运行情况。

Docker 提供了–cpu-period、–cpu-quota 两个参数来控制容器可以分配到的 CPU 时钟周期。–cpu-period 用来指定容器对 CPU 的使用要在多长时间内做一次重新分配，而–cpu-quota 用来指定在这个周期内，最多可以有多少时间用来跑这个容器。跟–cpu-shares 不同的是，这种配置是指定一个绝对值，而且没有弹性在里面，容器对 CPU 资源的使用绝对不会超过配置的值。

–cpu-period 和–cpu-quota 的单位为微秒（μs）。–cpu-period 的最小值为 1 000 μs，最大值为 1 s（10^6 μs），默认值为 0.1 s（100 000 μs）。–cpu-quota 的值默认为–1，表示不做控制。

举个例子，如果容器进程每 1 s 需要使用单个 CPU 的 0.2 s 时间，可以将 –cpu-period 设置为 1 000 000（即 1 s），将 –cpu-quota 设置为 200 000（0.2 s）。当然，在多核情况下，如果允许容器进程需要完全占用两个 CPU，则可以将 –cpu-period 设置为 100 000（即 0.1 s），将 –cpu-quota 设置为 200 000（0.2 s）。

使用示例：

使用命令 docker run -tid –cpu-period 100000 –cpu-quota 200000 ubuntu 创建容器，则最终生成的 Cgroups 的 CPU 周期配置可以在下面的文件中找到：

```
root@ubuntu:~# cat /sys/fs/Cgroups/cpu/docker/<容器的完整 ID>/cpu.cfs_period_us
100000
root@ubuntu:~# cat /sys/fs/Cgroups/cpu/docker/<容器的完整长 ID>/cpu.cfs_quota_us
200000
```

对于多核 CPU 的服务器，Docker 还可以控制容器运行时限定使用哪些 CPU 内核和内存节点，即使用–cpuset-cpus 和–cpuset-mems 参数。对具有 NUMA 拓扑（具有多 CPU、多内存节点）的服务器尤其有用，可以对需要高性能计算的容器进行性能最优的配置。如

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

果服务器只有一个内存节点，则–cpuset-mems 的配置基本上不会有明显效果。

使用示例：

```
# docker run -tid --name 1daoyun --cpuset-cpus 0,1 ubuntu:14.04.3
d566f10cd54c5c5da1ba908777fb814eb32327cbfb5bdd2f9ed0d5251cd81a17
[root@registry system.slice]# ll
total 0
-rw-r--r-- 1 root root 0 Feb 25 19:42 Cgroups.clone_children
--w--w--w- 1 root root 0 Feb 25 19:42 Cgroups.event_control
-rw-r--r-- 1 root root 0 Feb 25 19:42 Cgroups.procs
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.cpu_exclusive
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.cpus
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.mem_exclusive
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.mem_hardwall
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.memory_migrate
-r--r--r-- 1 root root 0 Feb 25 19:42 cpuset.memory_pressure
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.memory_spread_page
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.memory_spread_slab
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.mems
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.sched_load_balance
-rw-r--r-- 1 root root 0 Feb 25 19:42 cpuset.sched_relax_domain_level
drwxr-xr-x 2 root root 0 Feb 25 20:21 docker-5440c0d87c42047da0132f9e10877a4e8582
d0b51c312a5c7b7e9b5c2fc3004d.scope
drwxr-xr-x 2 root root 0 Feb 25 19:42 docker-68057601b59a1e4a76db76aa7c0ec4600
fea11df30790c136bff7f52e19e0f7d.scope
drwxr-xr-x 2 root root 0 Feb 26 19:49 docker-d566f10cd54c5c5da1ba908777fb814eb
32327cbfb5bdd2f9ed0d5251cd81a17.scope
-rw-r--r-- 1 root root 0 Feb 25 19:42 notify_on_release
-rw-r--r-- 1 root root 0 Feb 25 19:42 tasks
[root@registry system.slice]# cat docker-d566f10cd54c5c5da1ba908777fb814eb32327
cbfb5bdd2f9ed0d5251cd81a17.scope/cpuset.cpus
0-1
```

项目实训

【实训题目】

1. 查看容器 web5 的进程 Pid，查看命名空间和命名空间 IP。
2. 创建一个 Cgroups，并把进程 ID 写入 tasks 文件。

【实训目的】

1. 掌握命名空间 NameSpace 的使用方法，会使用命名空间查看容器的各种信息。
2. 掌握 Cgroups 的使用方法，能通过 Cgroups 管理。

【实训内容】

1. 查看容器的进程 Pid。
2. 查看命名空间。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

3. 查看命名空间 IP。
4. 创建一个 Cgroups，在 Cgroups 目录下的子系统中创建一个目录。
5. 查看这个目录，会发现目录下自动创建了控制文件。
6. 把进程 ID 写到 tasks 文件。

单元小结

本单元主要介绍了容器的底层技术，包括存储、网络、API 等。在介绍一个大的项目时，学生可能无法很快速地全部掌握，因此把一个大的项目分成几个模块，然后一一去介绍，这对于这个大项目的学习是有很大帮助的。介绍完本单元，学生对容器的运维又有了更高深的认识，对以后的学习或工作都有很大的帮助。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

单元 5 Docker 容器编排与集群

学习目标

知识目标

- 了解容器编排的管理方法。
- 掌握容器编排的基本使用。
- 了解容器集群的管理方法。
- 掌握容器集群的基本使用。

技能目标

- 掌握容器编排与集群的安装。
- 掌握容器编排与集群的基本管理与使用。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

学习情境

随着容器使用得越发频繁，应用服务和容器间的关系更是复杂，面对这种情况，研发部希望能使用更好的方法管理这些服务和对应的容器。工程师小魏经过分析之后，决定使用集群来统一管理应用服务。最近公司有个项目，需要将前台 Web 服务器和后台数据库，再加上负载均衡容器统一部署在容器中，研发部把这个项目交给了工程师小魏，面对这个任务，小魏决定使用 Docker-compose 容器编排服务，通过一个单独的 docker-compose.yml 模板文件（YAML 格式）为一个项目来定义一组相关联的应用容器。

任务 5.1　Docker-compose 容器编排的使用

任务描述

1. 安装 Docker-compose。
2. 构建 Python Web 镜像和容器。
3. 使用 Docker-compose。

知识学习

1. Compose 简介

Compose 的定位是“定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker Applications）”，其前身是开源项目 Fig，目前仍然兼容 Fig 格式的模板文件。

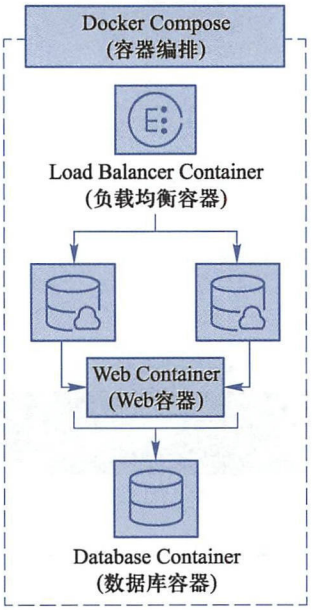


图 5-1-1
Web 项目架构示意图

通过之前的介绍可以知道，使用一个 Dockerfile 模板文件可以让用户很方便地定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身外，往往还需要加上后端的数据库服务容器，甚至还包括负载均衡容器等。图 5-1-1 所示为 Web 项目架构示意图。

Compose 恰好满足了这样的需求。它允许用户通过一个单独的 docker-compose.yml 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目。

2. Compose 中的 service 和 project

Compose 中有两个重要的概念。

① 服务（service）：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。

② 项目（project）：由一组关联的应用容器组成的一个完整业务单元，在 docker-compose.yml 文件中定义。

Compose 的默认管理对象是项目，可通过子命令对项目中的一组容器进行便捷的生命周期管理。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

Compose 项目由 Python 编写，在实现上调用了 Docker 服务提供的 API 来对容器进行管理。因此，只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

任务实施

1. 安装 Compose

Compose 目前支持 Linux 和 Mac OS 平台，两者的安装过程大同小异。

安装 Compose 之前，要先安装 Docker（需要 Docker Engine 1.7.1+），可参考之前的单元，此外不再赘述。

Compose 可以通过 Python 的 pip 工具进行安装，也可以直接下载编译好的二进制文件进行使用，甚至可以直接运行在 Docker 容器中。

前两种方式是传统方式，适合在本地环境下安装使用；最后一种方式则不破坏系统环境，更适合云计算场景。

① 安装 Docker Compose（选择 1.8.1 版本）。

```
# yum -y install epel-release
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: mirrors.cn99.com
 * extras: mirrors.cn99.com
 * updates: mirrors.cn99.com
Resolving Dependencies
--> Running transaction check
---> Package epel-release.noarch 0:7-9 will be installed
--> Finished Dependency Resolution

# yum -y install python-pip

# sudo pip install -U docker-compose
[root@docker bin]# sudo pip install -U docker-compose
Collecting docker-compose
  Downloading docker_compose-1.11.1-py2.py3-none-any.whl (85kB)
    100% |████████████████████████████████| 92kB 157kB/s
Collecting cached-property<2,>=1.2.0 (from docker-compose)
  Downloading cached_property-1.3.0-py2.py3-none-any.whl
Collecting backports.ssl-match-hostname>=3.5; python_version < "3.5" (from docker-compose)
  Downloading backports.ssl_match_hostname-3.5.0.1.tar.gz
Collecting ipaddress>=1.0.16; python_version < "3.3" (from docker-compose)
```

② 安装好以后更新 pip 工具。

```
# pip install --upgrade pip
Collecting pip
  Downloading pip-9.0.1-py2.py3-none-any.whl (1.3MB)
    100% |████████████████████████████████| 1.3MB 28kB/s
Installing collected packages: pip
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
  Found existing installation: pip 8.1.2
    Uninstalling pip-8.1.2:
      Successfully uninstalled pip-8.1.2
Successfully installed pip-9.0.1
```

③ 验证 Docker-compose 是否可用（注意要 yum clean all）。

```
# sudo docker-compose --version
docker-compose version 1.8.1, build 258538
```

④ 离线安装。

首先下载 [docker-compose-Linux-x86_64]，地址为https://github.com/docker/compose/releases/download/1.8.1/docker-compose-Linux-x86_64，然后重新命名并添加可执行权限：

```
[root@docker bin]# mv docker-compose-Linux-x86_64 /usr/local/bin/docker-compose;
[root@docker bin]# chmod +x /usr/local/bin/docker-compose
[root@docker bin]# docker-compose --version
docker-compose version 1.8.1, build 878cff1
```

2. 应用 Compose 的案例

编写一个 Python Web 案例，应用 Compose 编排服务。案例包含两个容器、一个 Python Web 容器、一个是 Redis 数据库容器。

下面首先构建 Python Web 镜像和容器。

① 创建 composeapp 目录。

```
# mkdir composeapp
# cd composeapp/
# touch Dockerfile
```

② 编写 Python 程序。

```
# touch app.py
# vi app.py
from flask import Flask
from redis import Redis
import os

app = Flask(__name__)
redis = Redis(host="redis", port=6379)

@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello Docker Book reader! I have been seen {0} times'.format(redis.get('hits'))

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

③ 创建 Python 的依赖包描述文件。

```
# touch requirements.txt
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
# vi requirements.txt
flask
redis
```

④ 编写 Python Web 容器的镜像 Dockerfile 文件。

```
# vi Dockerfile
FROM python:2.7
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2016-08-01

ADD . /composeapp

WORKDIR /composeapp

RUN pip install -r requirements.txt
```

⑤ 构建镜像。

```
# docker build -t jamtur01/composeapp .
Sending build context to Docker daemon 4.096 kB
Step 1 : FROM python:2.7
Trying to pull repository 192.168.200.11:5000/python ...
Pulling repository 192.168.200.11:5000/python
Error: image python not found
Trying to pull repository docker.io/library/python ...
2.7: Pulling from docker.io/library/python
75a822cd7888: Pull complete
57de64c72267: Pull complete
4306be1e8943: Pull complete
871436ab7225: Pull complete
37c937b0ca47: Pull complete
608a51124afe: Pull complete
086c59e7b25f: Pull complete
Digest: sha256:43b6a6df624798dd8889bde2c02a48a263faa1d72f0de9e134de84c4079ac365
Status: Downloaded newer image for docker.io/python:2.7
 ---> acf0d719f268
Step 2 : MAINTAINER James Turnbull <james@example.com>
 ---> Running in 19e30769bcd3
 ---> 6efea2b07fec
Removing intermediate container 19e30769bcd3
Step 3 : ENV REFRESHED_AT 2016-08-01
 ---> Running in 1c55d12d766b
 ---> fdf17727b383
Removing intermediate container 1c55d12d766b
Step 4 : ADD . /composeapp
 ---> 96618268b61c
Removing intermediate container cd7b0ed6e203
Step 5 : WORKDIR /composeapp
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
 ---> Running in 5f04f2e5adca
 ---> c504ecdffde0
Removing intermediate container 5f04f2e5adca
Step 6 : RUN pip install -r requirements.txt
 ---> Running in 7433546dcc5c
Collecting flask (from -r requirements.txt (line 1))
  Downloading Flask-0.12-py2.py3-none-any.whl (82kB)
Collecting redis (from -r requirements.txt (line 2))
  Downloading redis-2.10.5-py2.py3-none-any.whl (60kB)
Collecting itsdangerous>=0.21 (from flask->-r requirements.txt (line 1))
  Downloading itsdangerous-0.24.tar.gz (46kB)
Collecting Jinja2>=2.4 (from flask->-r requirements.txt (line 1))
  Downloading Jinja2-2.8-py2.py3-none-any.whl (263kB)
Collecting Werkzeug>=0.7 (from flask->-r requirements.txt (line 1))
  Downloading Werkzeug-0.11.13-py2.py3-none-any.whl (308kB)
Collecting click>=2.0 (from flask->-r requirements.txt (line 1))
  Downloading click-6.6-py2.py3-none-any.whl (71kB)
Collecting MarkupSafe (from Jinja2>=2.4->flask->-r requirements.txt (line 1))
  Downloading MarkupSafe-0.23.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous: started
  Running setup.py bdist_wheel for itsdangerous: finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/fc/a8/66/24d655233c757e178d45dea2de22
a04c6d92766abfb741129a
  Running setup.py bdist_wheel for MarkupSafe: started
  Running setup.py bdist_wheel for MarkupSafe: finished with status 'done'
  Stored in directory: /root/.cache/pip/wheels/a3/fa/dc/0198eed9ad95489b8a4f45d14dd5
d2aee3f8984e46862c5748
Successfully built itsdangerous MarkupSafe
Installing collected packages: itsdangerous, MarkupSafe, Jinja2, Werkzeug, click, flask, redis
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.11.13 click-6.6 flask-0.12
itsdangerous- 0.24 redis-2.10.5
 ---> e031fec32547
Removing intermediate container 7433546dcc5c
Successfully built e031fec32547
```

⑥ 以 docker run 操作运行案例。

```
[root@client composeapp]# sudo docker run-d -p 5000:5000 -v.:/composeapp--link redis:redis
--name jamtur01/composeapp python app.py
Unable to find image 'python:latest' locally
Trying to pull repository 192.168.200.11:5000/python ...
Pulling repository 192.168.200.11:5000/python
Error: image python not found
Trying to pull repository docker.io/library/python ...
latest: Pulling from docker.io/library/python
75a822cd7888: Already exists
57de64c72267: Already exists
```

版权所有，严禁传播，违者自负法律责任！


```
    4306be1e8943: Already exists
    871436ab7225: Already exists
    37c937b0ca47: Already exists
    2172fb7edb3a: Pull complete
    dcec4e4061ab: Pull complete
    Digest: sha256:1fa4d0110c7093fdb1306b4b290824a37169ca6a19f091b7227967647726169a
    Status: Downloaded newer image for docker.io/python:latest
    docker: Error response from daemon: Invalid container name (jamtur01/composeapp), only
[a-zA-Z0-9] [a-zA-Z0-9_.-] are allowed.
    See '/usr/bin/docker-current run --help'.
```

⑦ 构建 Docker Compose yml 文件，即 docker-compose.yml 文件：

```
    # vi docker-compose.yml
    web:
      image: jamtur01/composeapp
      command: python app.py
      ports:
       - "5000:5000"
      volumes:
       - .:/composeapp
      links:
       - redis
    redis:
      image: redis
```

⑧ 切换到 composeapp 目录下，以 compose 的方式运行：

```
    # docker-compose up
    Pulling redis (redis:latest)...
    Trying to pull repository 192.168.200.18:5000/redis ...
    Pulling repository 192.168.200.18:5000/redis
    Error: image redis not found
    Trying to pull repository docker.io/library/redis ...
    latest: Pulling from docker.io/library/redis
    75a822cd7888: Already exists
    e40c2fafe648: Pull complete
    ce384d4aea4f: Pull complete
    5e29dd684b84: Pull complete
    29a3c975c335: Pull complete
    a405554540f9: Pull complete
    4b2454731fda: Pull complete
    Digest: sha256:be03cb3b2e0d290c3d6670d1843b6036c1914663574bb624ffb62535344de5b4
    Status: Downloaded newer image for docker.io/redis:latest
    Creating composeapp_redis_1
    Creating composeapp_web_1
    Attaching to composeapp_redis_1, composeapp_web_1
    redis_1   | 1:C 01 Jan 12:35:45.005 # Warning: no config file specified, using the default config.
In order to specify a config file use redis-server /path/to/redis.conf
```

```
redis_1  |                 _.-_
redis_1  |             _.-``__''-._
redis_1  |        _.-``    `.  `_.  ''-._           Redis 3.2.6 (00000000/0) 64 bit
redis_1  |    .-`` .-```.  ```\/    _.,_ ''-._
redis_1  |   (    '      ,       .-`  | `,    )     Running in standalone mode
redis_1  |   |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
redis_1  |   |    `-._   `._    /     _.-'    |     PID: 1
redis_1  |    `-._    `-._  `-./  _.-'    _.-'
redis_1  |   |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1  |   |    `-._`-._        _.-'_.-'    |           http://redis.io
redis_1  |    `-._    `-._`-.__.-'_.-'    _.-'
redis_1  |   |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1  |   |    `-._`-._        _.-'_.-'    |
redis_1  |    `-._    `-._`-.__.-'_.-'    _.-'
redis_1  |        `-._    `-.__.-'    _.-'
redis_1  |            `-._        _.-'
redis_1  |                `-.__.-'
redis_1  |
redis_1  | 1:M 01 Jan 12:35:45.008 # WARNING: The TCP backlog setting of 511 cannot be enforced because/proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1  | 1:M 01 Jan 12:35:45.008 # Server started, Redis version 3.2.6
redis_1  | 1:M 01 Jan 12:35:45.008 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
redis_1  | 1:M 01 Jan 12:35:45.008 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
redis_1  | 1:M 01 Jan 12:35:45.008 * The server is now ready to accept connections on port 6379
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
web_1    |  * Restarting with stat
web_1    |  * Debugger is active!
web_1    |  * Debugger pin code: 457-652-386
web_1    | 192.168.0.51 - - [01/Jan/2017 12:37:42] "GET / HTTP/1.1" 200 -
web_1    | 192.168.0.51 - - [01/Jan/2017 12:37:42] "GET /favicon.ico HTTP/1.1" 404 -
```

⑨ 最后在网页上验证效果，如图 5-1-2 所示。

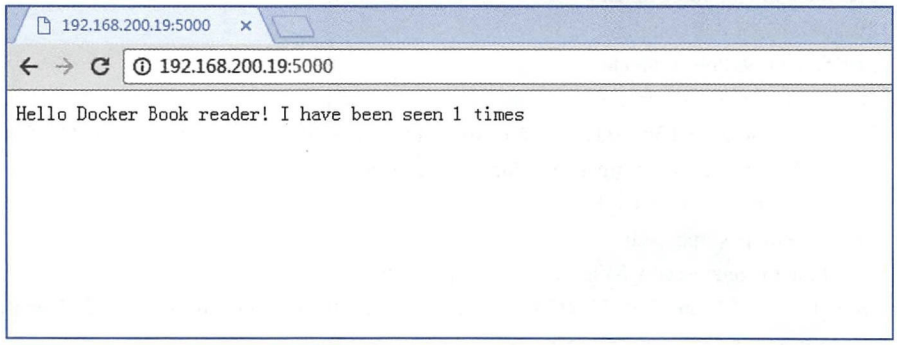


图 5-1-2
实例网页验证效果

项目实训

笔 记

【实训题目】

在 CentOS 系统上安装 Compose，编写一个 Python Web 案例并用 docker-compose 运行案例。

【实训目的】

1. 掌握 Compose 的安装。
2. 掌握 Compose 的应用方法。

【实训内容】

1. 在 CentOS 系统上安装 Compose。
2. 验证 Compose 是否安装正确。
3. 构建 Python Web 镜像和容器。
4. 以 docker-compose 运行案例。
5. 验证是否安装成功。

任务 5.2　Docker Swarm 集群的配置与管理

任务描述

1. 安装 Docker Swarm。
2. 在线创建 Swarm 集群和自动编配。
3. 离线创建 Swarm 集群和自动编配。

知识学习

1. Docker Swarm 简介

Swarm 这个项目名称特别贴切。在 Wiki 的解释中，Swarm behavior 是指动物的群集行为。比如，人们常见的蜂群、鱼群、秋天往南飞的雁群都可以称作 Swarm behavior。

Docker Swarm 是 Docker 公司官方在 2014 年 12 月初发布的一套管理 Docker 集群的工具。它将一群 Docker 宿主机变成一个单一的、虚拟的主机。Swarm 使用标准的 Docker API 接口作为其前端访问入口，换言之，各种形式的 Docker 工具（如 Dokku、Compose、Krane、Deis、docker-py 和 Docker 本身等）都可以很容易地与 Swarm 进行集成，Swarm 结构图如图 5-2-1 所示。

在使用 Swarm 管理 Docker 集群时，会有一个 swarm manager 及若干的 swarm node，swarm manager 上运行 swarm daemon，用户只需要与 swarm manager 通信即可，然后 swarm manager 根据 discovery service 的信息选择一个 swarm node 来运行 container。

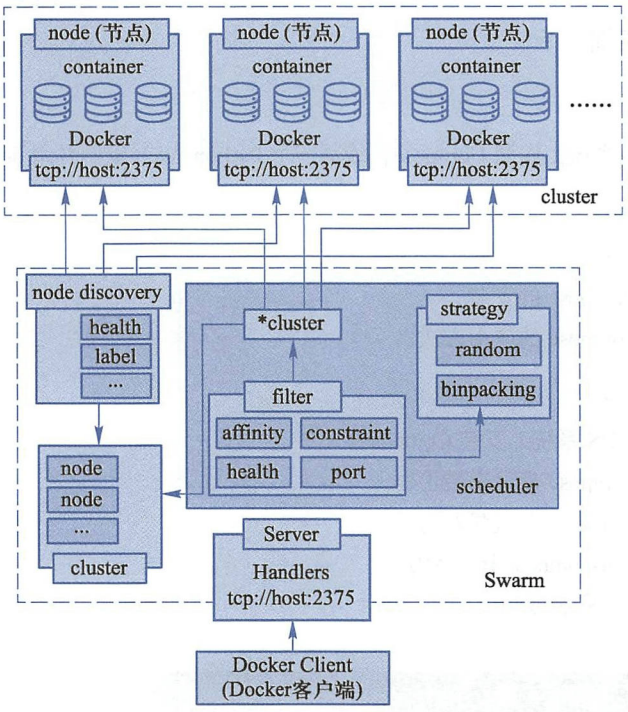


图 5-2-1
Swarm 结构图

值得注意的是，swarm daemon 只是一个任务调度器（scheduler）和路由器（router），它本身不运行容器，它只接收 Docker client 发送过来的请求，调度合适的 swarm node 来运行 container。这意味着，即使 swarm daemon 由于某些原因挂掉了，已经运行起来的容器也不会有任何影响。

2. Docker Swarm 特点

① Swarm 对外以 Docker API 接口呈现，这样带来的好处是，如果现有系统使用 Docker Engine，则可以平滑地将 Docker Engine 切到 Swarm 上，无须改动现有系统。

② Swarm 对用户来说，之前使用 Docker 的经验可以继承过来，非常容易上手，学习成本和二次开发成本都比较低。同时，Swarm 本身专注于 Docker 集群管理，非常轻量，占用资源也非常少。Batteries included but swappable，简单来说，就是插件化机制，Swarm 中的各个模块都抽象出了 API，可以根据自己的特点进行定制实现。

③ Swarm 自身对 Docker 命令参数支持得比较完善，Swarm 目前与 Docker 是同步发布的。Docker 的新功能都会第一时间在 Swarm 中体现。

任务实施

1. 安装 Docker Swarm 的环境

有以下两点需要注意。

第一，集群中的每台节点上面的 Docker 的版本都不能小于 1.4。

第二，swarm manager 能够与每台 swarm node 进行通信，集群中的每台节点的 Docker daemon 都必须监听同一个网络接口。

① 在 smoker 和 joker 节点手动安装 docker-engine-1.7.0-1。

```
# ls
docker-engine-1.7.0-1.el7.CentOS.x86_64.rpm
# yum -y install docker-engine-1.7.0-1.el7.CentOS.x86_64.rpm
Loaded plugins: fastestmirror
Examining docker-engine-1.7.0-1.el7.CentOS.x86_64.rpm: docker-engine-1.7.0-1.el7.CentOS.x86_64
Marking docker-engine-1.7.0-1.el7.CentOS.x86_64.rpm to be installed
Resolving Dependencies
--> Running transaction check
---> Package docker-engine.x86_64 0:1.7.0-1.el7.CentOS will be installed
--> Processing Dependency: libc.so.6 for package: docker-engine-1.7.0-1.el7.CentOS.x86_64
  Installing : docker-engine-1.7.0-1.el7.CentOS.x86_64                    8/8
  Verifying  : nss-softokn-freebl-3.16.2.3-14.4.el7.i686                  6/8
  Verifying  : readline-6.2-9.el7.i686                                    7/8
  Verifying  : libgcc-4.8.5-11.el7.i686
Installed:
  docker-engine.x86_64 0:1.7.0-1.el7.CentOS
Dependency Installed:
  glibc.i686 0:2.17-157.el7_3.1          libgcc.i686 0:4.8.5-11.el7
  libstdc++.i686 0:4.8.5-11.el7          ncurses-libs.i686 0:5.9-13.20130511.el7
  nss-softokn-freebl.i686 0:3.16.2.3-14.4.el7  readline.i686 0:6.2-9.el7
  sqlite.i686 0:3.7.17-8.el7
Complete!
```

② 验证 Docker 版本信息。

```
# docker version
Client version: 1.7.0
Client API version: 1.19
Go version (client): go1.4.2
Git commit (client): 0baf609
OS/Arch (client): linux/amd64
Get http:///var/run/docker.sock/v1.19/version: dial unix /var/run/docker.sock: no such file or
directory. Are you trying to connect to a TLS-enabled daemon without TLS?
```

③ 建立主机映射关系。

```
# vi /etc/hosts
127.0.0.1    localhost localhost.localdomain localhost4 localhost4.localdomain4
::1          localhost localhost.localdomain localhost6 localhost6.localdomain6
192.168.200.37 smoker
192.168.200.38 joker
[root@localhost yum.repos.d]# systemctl enable docker.service
```

④ 在 smoker 和 joker 节点下载 Swarm 镜像。

```
# docker pull swarm
latest: Pulling from swarm
4f9efaf672f5: Pull complete
fb1b1184d0c7: Pull complete
21e02dd8277b: Pull complete
```

```
7a01afb8e091: Pull complete
e083d0961d7f: Pull complete
bbf3c6645e31: Pull complete
6df8cd26dd3c: Pull complete
23fc6f8ade4e: Pull complete
Digest: sha256:e1c8c519de006d41043dcdb93b61c01ab512a90f88f673ad5b63b76d9f87c7ff
Status: Downloaded newer image for swarm:latest
```

⑤ 验证本地 images。

```
[root@localhost yum.repos.d]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
swarm               latest              23fc6f8ade4e        4 months ago        19.47 MB
```

⑥ 在 docker.service 服务的配置文件里加入端口监听，实现监听同一端口。需要加入的代码为：

```
    ExecStart=/usr/bin/docker-current daemon --registry-mirror=http://ef0cb1d0.m.daocloud.io -H
0.0.0.0: 2375 -H unix:///var/run/docker.sock \
    # vi /usr/lib/systemd/system/docker.service
    [Unit]
    Description=Docker Application Container Engine
    Documentation=http://docs.docker.com
    After=network.target rhel-push-plugin.socket
    Wants=docker-storage-setup.service

    [Service]
    Type=notify
    NotifyAccess=all
    EnvironmentFile=-/etc/sysconfig/docker
    EnvironmentFile=-/etc/sysconfig/docker-storage
    EnvironmentFile=-/etc/sysconfig/docker-network
    Environment=GOTRACEBACK=crash
    ExecStart=/usr/bin/docker-current daemon --registry-mirror=http://ef0cb1d0.m.daocloud.io -H
0.0.0.0: 2375 -H unix:///var/run/docker.sock \
            --exec-opt native.Cgroupsdriver=systemd \
            $OPTIONS \
            $DOCKER_STORAGE_OPTIONS \
            $DOCKER_NETWORK_OPTIONS \
            $ADD_REGISTRY \
            $BLOCK_REGISTRY \
            $INSECURE_REGISTRY
    LimitNOFILE=1048576
    LimitNPROC=1048576
    LimitCORE=infinity
    TimeoutStartSec=0
    MountFlags=slave
```

其中，第 2 行代码中的 2375 为共同监听的端口，可变。

⑦ 验证端口监听状态。

```
# netstat -anpl | grep 2375
tcp6       0      0 :::2375                 :::*
```

2. 在线创建 Swarm 集群和自动编排

① 在 smoker 节点创建集群令牌。

```
# hostnamectl set-hostname smoker
# docker run --rm swarm create
e97d905a9744006cff0f44aa7cfe57dd
```

注意

这个 ID 的值将会是每台主机加入验证的令牌。

② 将 smoker 节点加入 Swarm 集群。

```
# docker run -d swarm join --addr=192.168.200.37:2375 token://e97d905a9744006cff0f44aa7cfe57dd
259d21ea8e9a6eee96036fec5f090379ad880cc8ccc9de5f739c8a45fcd97e55
```

③ 查看 smoker 节点 logs。

```
# docker logs 259d21ea8e9a6eee96036fec5f090379ad880cc8ccc9de5f739c8a45fcd97e55
time="2017-01-09T03:04:52Z" level=info msg="Registering on the discovery service every
1m0s..." addr="192.168.200.37:2375" discovery="token://e97d905a9744006cff0f44aa7cfe57dd"
time="2017-01-09T03:05:56Z" level=info msg="Registering on the discovery service every
1m0s..." addr="192.168.200.37:2375" discovery="token://e97d905a9744006cff0f44aa7cfe57dd"
time="2017-01-09T03:06:57Z" level=info msg="Registering on the discovery service every
1m0s..." addr="192.168.200.37:2375" discovery="token://e97d905a9744006cff0f44aa7cfe57dd"
```

可以看到，每隔 1 min 就会向发现服务进行注册，告诉后端 Docker Hub 该节点可用，该 Docker 服务器也可以被使用。

④ 列出 Swarm 节点。

```
# docker run --rm swarm list token://e97d905a9744006cff0f44aa7cfe57dd
192.168.200.38:2375
192.168.200.37:2375
```

⑤ 在 joker 节点加入 Swarm 集群。

```
# docker run -d swarm join --addr=192.168.200.38:2375 token://e97d905a9744006cff0f44aa7cfe57dd
4d55041343c32432810333c3af70f4c367035644cab447189def75cb17753206
# docker logs 4d55041343c32432810333c3af70f4c367035644cab447189def75cb17753206
time="2017-01-09T03:06:03Z" level=info msg="Registering on the discovery service every
1m0s..." addr="192.168.200.38:2375" discovery="token://e97d905a9744006cff0f44aa7cfe57dd"
time="2017-01-09T03:07:04Z" level=info msg="Registering on the discovery service every
1m0s..." addr="192.168.200.38:2375" discovery="token://e97d905a9744006cff0f44aa7cfe57dd"
time="2017-01-09T03:08:05Z" level=info msg="Registering on the discovery service every
1m0s..." addr="192.168.200.38:2375" discovery="token://e97d905a9744006cff0f44aa7cfe57dd"
```

⑥ 在 smoker 启动集群管理者。

```
[root@localhost opt]# docker run -d -p 2380:2375 swarm manage token:// e97d905a9744006cff0 f44aa7cfe57dd
22aaa295d5b54c42baaf7099997755af6cf8d7e6f8f1e04fd5b42be5a2a95174
```

⑦ 在 Swarm 集群中运行 docker info。

```
# docker -H tcp://manager:2380 info
Containers: 0
Images: 0
Role: primary
Strategy: spread
Filters: health, port, containerslots, dependency, affinity, constraint, whitelist
Nodes: 2
(unknown): smoker:2375
  └ ID:
  └ Status: Pending
  └ Containers: 0
  └ Reserved CPUs: 0 / 0
  └ Reserved Memory: 0 B / 0 B
  └ Labels:
  └ UpdatedAt: 2017-01-23T09:13:37Z
  └ ServerVersion:
 (unknown): joker:2375
  └ ID:
  └ Status: Pending
  └ Containers: 0
  └ Reserved CPUs: 0 / 0
  └ Reserved Memory: 0 B / 0 B
  └ Labels:
  └ UpdatedAt: 2017-01-23T09:13:37Z
  └ ServerVersion:
Kernel Version: 3.10.0-514.6.1.el7.x86_64
Operating System: linux
CPUs: 0
Total Memory: 0 B
Name: 79aa67e26d1f
```

⑧ 循环创建 6 个 Nginx 容器，实现自动编配。

```
# for i in 'seq 1 6';do sudo docker -H tcp://localhost:2380 run -d --name www-$i -p 80 busybox;done
```

⑨ 验证自动编配结果，查看 Nginx 容器在集群里分布的位置。

```
# docker -H 192.168.200.141:2377 ps -a
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
7daec88a333f   swarm   "/swarm   2 minutes   Up 2   192.168.200.   node1/clever_bhaskara
```


非卖品，仅供非商业用途或交流学习使用

```
                                manage    ago              minutes          141:2377->
                                file:/"                                     2375/tcp
6317900300ea  busybox           "sh"      10 minutes ago   Up 10 minutes                node1/determined_saha
3133e82fd180  busybox           "sh"      10 minutes ago   Up 10 minutes                node2/grave_engelbart
0c8421482e12  busybox           "sh"      11 minutes ago   Up 11 minutes                node1/loving_kilby
f51e54ab8f00  busybox           "sh"      11 minutes ago   Up 11 minutes                node2/high_booth
b8d452b6da6a  busybox           "sh"      12 minutes ago   Up 12 minutes                node2/focused_hodgkin
643d5d7adad9  swarm             "/swarm   14 minutes ago   Up 14 minutes
                                manage file:/"
```

3. 离线创建 Swarm 集群和自动编排

离线 Docker Swarm 环境安装和在线安装基本一样，只是安装包要上传到服务器上，也是监听同一网络端口。

① 更改防火墙设置。

```
# hostnamectl set-hostname node1
# iptables -F
# iptables -X
# iptables -Z
# /usr/sbin/iptables-save
# Generated by iptables-save v1.4.21 on Tue Jan 24 18:54:44 2017
*filter
:INPUT ACCEPT [36:2480]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [21:1948]
COMMIT
# Completed on Tue Jan 24 18:54:44 2017
# vi /etc/sysctl.conf
# System default settings live in /usr/lib/sysctl.d/00-system.conf.
# To override those settings, enter new settings here, or in an /etc/sysctl.d/<name>.conf file
# For more information, see sysctl.conf(5) and sysctl.d(5).
net.ipv4.ip_forward = 1
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
"/etc/sysctl.conf" 8L, 318C written
 # sysctl -p
net.ipv4.ip_forward = 1
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
```

② 安装 Docker 组件。

```
# yum install -y docker-io
Loaded plugins: fastestmirror
CentOS                                                    | 3.6 kB   00:00:00
iaas                                                      | 2.9 kB   00:00:00
--> Processing Dependency: libdevmapper-event.so.1.02()(64bit) for package: 7:lvm2-2.02.130-
5.el7_2.5.x86_64
```


非卖品，仅供非商业用途或交流学习使用

```
Installed:
  docker.x86_64 0:1.10.3-46.el7.CentOS.14

Dependency Installed:
  device-mapper-event.x86_64 7:1.02.107-5.el7_2.5
  device-mapper-event-libs.x86_64 7:1.02.107-5.el7_2.5
  device-mapper-persistent-data.x86_64 0:0.6.2-1.el7_2
  docker-common.x86_64 0:1.10.3-46.el7.CentOS.14
  docker-selinux.x86_64 0:1.10.3-46.el7.CentOS.14
  libaio.x86_64 0:0.3.109-13.el7
  libseccomp.x86_64 0:2.2.1-1.el7
  lvm2.x86_64 7:2.02.130-5.el7_2.5
  lvm2-libs.x86_64 7:2.02.130-5.el7_2.5
  oci-register-machine.x86_64 1:0-1.8.gitaf6c129.el7
  oci-systemd-hook.x86_64 1:0.1.4-4.git41491a3.el7
  yajl.x86_64 0:2.0.4-4.el7
Complete!
```

③ 启动 Docker 服务，并检查服务状态。

```
# systemctl start docker
# systemctl status docker
 docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: active (running) since Tue 2017-01-24 19:04:48 UTC; 10s ago
     Docs: http://docs.docker.com
 Main PID: 1212 (docker-current)
   Cgroups: /system.slice/docker.service
            └─1212 /usr/bin/docker-current daemon --exec-opt native.Cgroupsdriver=systemd --selinux-e...

Jan 24 19:04:46 node1 docker-current[1212]: time="2017-01-24T19:04:46.334911559Z" level=info msg...ds"
Jan 24 19:04:46 node1 docker-current[1212]: time="2017-01-24T19:04:46.431782324Z" level=warning msg...
Jan 24 19:04:46 node1 docker-current[1212]: time="2017-01-24T19:04:46.545914706Z" level=info msg...se"
Jan 24 19:04:46 node1 docker-current[1212]: time="2017-01-24T19:04:46.887056097Z" level=info msg...ss"
Jan 24 19:04:48 node1 docker-current[1212]: time="2017-01-24T19:04:48.233891205Z" level=info msg...t."
Jan 24 19:04:48 node1 docker-current[1212]: time="2017-01-24T19:04:48.233987596Z" level=info msg...e."
Jan 24 19:04:48 node1 docker-current[1212]: time="2017-01-24T19:04:48.234012342Z" level=info msg...on"
Jan 24 19:04:48 node1 docker-current[1212]: time="2017-01-24T19:04:48.234043489Z" level=info msg...0.3
Jan 24 19:04:48 node1 systemd[1]: Started Docker Application Container Engine.
Jan 24 19:04:48 node1 docker-current[1212]: time="2017-01-24T19:04:48.276564234Z" level=info msg...ck"
Hint: Some lines were ellipsized, use -l to show in full.
```

④ 修改配置文件，监听同一端口号。

```
# vi /usr/lib/systemd/system/docker.service
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=network.target rhel-push-plugin.socket
```


非卖品，仅供非商业用途或交流学习使用

```
Wants=docker-storage-setup.service

[Service]
Type=notify
NotifyAccess=all
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
Environment=GOTRACEBACK=crash
ExecStart=/usr/bin/docker-current daemon -H 0.0.0.0:2375 -H unix:///var/run/docker.sock \
          --exec-opt native.Cgroupsdriver=systemd \
          $OPTIONS \
          $DOCKER_STORAGE_OPTIONS \
          $DOCKER_NETWORK_OPTIONS \
          $ADD_REGISTRY \
          $BLOCK_REGISTRY \
          $INSECURE_REGISTRY
LimitNOFILE=1048576
LimitNPROC=1048576
LimitCORE=infinity
TimeoutStartSec=0
MountFlags=slave
Restart=on-abnormal
"/usr/lib/systemd/system/docker.service" 30L, 843C written
```

修改配置以后需要重启服务。

```
# systemctl daemon-reload
# systemctl restart docker
# netstat -anpl | grep 2375
tcp6       0      0 :::2375                 :::*                    LISTEN      1371/docker-current
    # sudo systemctl restart docker
```

⑤ 下载 Swarm 组件。

```
# docker pull swarm
Using default tag: latest
Trying to pull repository docker.io/library/swarm ...
latest: Pulling from docker.io/library/swarm
ebe0176dcf9a: Pull complete
19f771faa982: Pull complete
902eeedf931a: Pull complete
Digest: sha256:2a66586181b8ffc169035e8462614ab48836bd578de51f2233fd20c86e7408cb
Status: Downloaded newer image for docker.io/swarm:latest
# cd /root
```

⑥ 创建集群，加入节点。

```
# vi cluster
20.0.0.47:2375
```


非卖品，仅供非商业用途或交流学习使用

```
20.0.0.48:2375
"cluster" [New] 2L, 30C written
```

⑦ 启动集群。

```
# docker run -d -p 2376:2375 -v $(pwd)/cluster:/tmp/cluster swarm manage file:///tmp/cluster
edcb5d5f2efbe22825b33265340be56bf508606891652f6026df61537b4ae439
```

⑧ 查询集群节点信息。

```
# docker run --rm -v $(pwd)/cluster:/tmp/cluster swarm list file:///tmp/cluster
20.0.0.47:2375
20.0.0.48:2375
```

⑨ 查询集群具体信息。

```
# docker -H 20.0.0.47:2376 info
Containers: 1
 Running: 1
 Paused: 0
 Stopped: 0
Images: 2
Server Version: swarm/1.2.6
Role: primary
Strategy: spread
Filters: health, port, containerslots, dependency, affinity, constraint, whitelist
Nodes: 2
 node1: 20.0.0.47:2375
  └ ID: 67EO:5EL4:J6GA:U5AN:W4YS:PKAB:INOH:KG7F:MO5D:25KF:SUM2:UNJ6
  └ Status: Healthy
  └ Containers: 1 (1 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 4.053 GiB
  └ Labels: kernelversion=3.10.0-327.el7.x86_64, operatingsystem=CentOS Linux 7 (Core),
storagedriver=devicemapper
  └ UpdatedAt: 2017-01-24T19:30:51Z
  └ ServerVersion: 1.10.3
 node2: 20.0.0.48:2375
  └ ID: EOIH:LT65:KRI4:F7OF:UNN3:X5X2:ZZJ5:NSTG:IFM7:HZVP:NO2N:Z7CL
  └ Status: Healthy
  └ Containers: 0 (0 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 4.053 GiB
  └ Labels: kernelversion=3.10.0-327.el7.x86_64, operatingsystem=CentOS Linux 7 (Core),
storagedriver=devicemapper
  └ UpdatedAt: 2017-01-24T19:30:59Z
  └ ServerVersion: 1.10.3
Plugins:
```


非卖品，仅供非商业用途或交流学习使用

```
 Volume:
 Network:
Kernel Version: 3.10.0-327.el7.x86_64
Operating System: linux
Architecture: amd64
Number of Docker Hooks: 2
CPUs: 2
Total Memory: 8.106 GiB
Name: edcb5d5f2efb
Registries:
```

⑩ 查询集群活动状态。

```
# docker ps -a
CONTAINER ID  IMAGE   COMMAND          CREATED      STATUS      PORTS                    NAMES
edcb5d5f2efb  swarm   "/swarm          3 minutes    Up 3        0.0.0.0:2376->2375/tcp   big_banach
                      manage file:/"   ago          minutes
```

⑪ 在集群内部署应用，并查询集群信息。

```
# docker -H 20.0.0.47:2376 run -idt busybox
53ffafc8a0fc3d60fa6c198d817a4869993509f8d25cf819cfee85936f27d25e
# docker -H 20.0.0.47:2376 info
Containers: 2
 Running: 2
 Paused: 0
 Stopped: 0
Images: 3
Server Version: swarm/1.2.6
Role: primary
Strategy: spread
Filters: health, port, containerslots, dependency, affinity, constraint, whitelist
Nodes: 2
 node1: 20.0.0.47:2375
  └ ID: 67EO:5EL4:J6GA:U5AN:W4YS:PKAB:INOH:KG7F:MO5D:25KF:SUM2:UNJ6
  └ Status: Healthy
  └ Containers: 1 (1 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 4.053 GiB
  └ Labels: kernelversion=3.10.0-327.el7.x86_64, operatingsystem=CentOS Linux 7 (Core), storagedriver=
devicemapper
  └ UpdatedAt: 2017-01-24T19:35:35Z
  └ ServerVersion: 1.10.3
 node2: 20.0.0.48:2375
  └ ID: EOIH:LT65:KRI4:F7OF:UNN3:X5X2:ZZJ5:NSTG:IFM7:HZVP:NO2N:Z7CL
  └ Status: Healthy
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
  └ Containers: 1 (1 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 4.053 GiB
  └ Labels: kernelversion=3.10.0-327.el7.x86_64, operatingsystem=CentOS Linux 7 (Core),
storagedriver=devicemapper
  └ UpdatedAt: 2017-01-24T19:36:02Z
  └ ServerVersion: 1.10.3
Plugins:
 Volume:
 Network:
Kernel Version: 3.10.0-327.el7.x86_64
Operating System: linux
Architecture: amd64
Number of Docker Hooks: 2
CPUs: 2
Total Memory: 8.106 GiB
Name: edcb5d5f2efb
Registries:
```

⑫ 查询集群内部容器活动状态。

```
# docker -H 20.0.0.47:2376 ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS                  NAMES
53ffafc8a0fc   busybox   "sh"                     About a minute ago   Up About a minute                  node2/trusting_goldwasser
edcb5d5f2efb   swarm     "/swarm manage file:/"   7 minutes ago    Up 7 minutes    20.0.0.47:2376->2375/tcp   node1/big_banach
```

⑬ 在集群内创建应用，实现自动部署。

```
# docker -H 20.0.0.47:2376 run -idt busybox
2c9bf4f83453ad58b5284ab7c374fd030666034a617d6b2d9d86cb09f6ab3459
# docker -H 20.0.0.47:2376 run -idt busybox
670ecb9bb4ce9e203e5e9e20875f5856c550c4ee0119ca371d9deff1ce53552b
# docker -H 20.0.0.47:2376 ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED              STATUS              PORTS                  NAMES
670ecb9bb4ce   busybox   "sh"                     6 seconds ago        Up 2 seconds                               node2/pensive_ptolemy
2c9bf4f83453   busybox   "sh"                     About a minute ago   Up 58 seconds                              node1/evil_allen
53ffafc8a0fc   busybox   "sh"                     2 minutes ago        Up 2 minutes                               node2/trusting_goldwasser
edcb5d5f2efb   swarm     "/swarm manage file:/"   9 minutes ago        Up 9 minutes        20.0.0.47:2376->2375/tcp   node1/big_banach
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

项目实训

笔 记

【实训题目】

在 CentOS 系统上安装 Docker Swarm，并使用在线和离线的方式创建 Swarm 集群和自动编排。

【实训目的】

1. 掌握 Docker Swarm 在 CentOS 系统上的安装。
2. 掌握 Swarm 集群的在线和离线创建。
3. 掌握 Swarm 集群的自动编排。

【实训内容】

1. 在 CentOS 系统上安装 Docker Swarm。
2. 在线创建 Swarm 集群和自动编排。
3. 离线创建 Swarm 集群和自动编排。

任务 5.3　Docker Consul 构建集群服务

任务描述

1. 创建目录来保存 Consul 的 Dockerfile。
2. 编写 Consul 的 Dockerfile 文件。
3. 配置 Consul.json 文件。
4. 构建 Consul 镜像。
5. 启动 Consul 服务，构建 Consul 集群。

知识学习

1. Docker Consul 简介

在 GitHub 上，Consul被称为“一种用于服务发现、监控和配置的工具”。Consul 是由 Vagrant 的开发公司 HashiCorp 开发的开源项目之一。它可以提供一个具有高可用性的分布式系统，用于注册服务、存储共享配置并保持多个数据中心的准确视图。此外，它是一个简单的 Go 程序，因此不需要部署。

2. Docker Consul 的实现原理

用户通过运行 Consul Agent 命令，可以在后台以守护进程的方式运行在所有 Consul 集群节点中。Consul Agent 命令以 Server 或者 Client 模式运行，并同时用 HTTP 或者 DNS 来运行检查和同步服务。Server 模式的 Agent 负责维护 Consul 集群状态，相应 RPC 查询，并且还要负责和其他数据中心进行 WAN Gossips 交换。Client 节点是相对无状态的，Client 的唯一活动就是转发请求给 Server 节点，以保持低延迟和少资源消耗。Docker Consul 是官网的一个典型系统结构，如图 5-3-1 所示。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

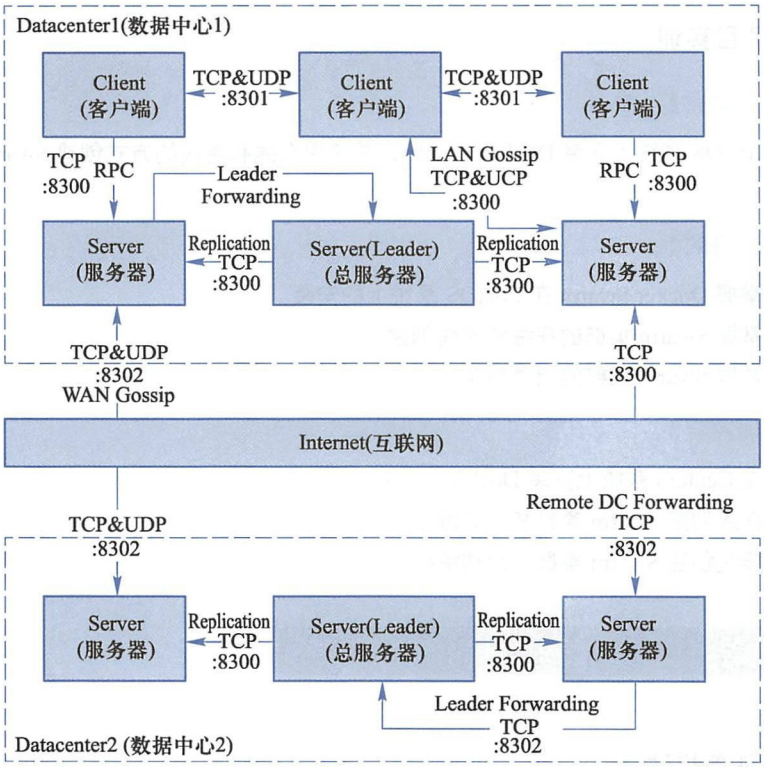


图 5-3-1
Docker Consul 系统架构图

Consul 建议每个 DataCenter 的 Server 节点在 3～5 个之间，以方便在失败以后进行数据恢复。Client 的数量可以任意。如图 5-3-1 所示，其中最重要的概念是 Gossip 协议。DataCenter 的所有节点都会参与到 Gossip 协议。Client 到 Server 会通过 LAN Gossip。所有的节点都在 Gossip pool 中，通过消息层来实现节点之间的通信以达到故障检测的目的，并且不需要给 Client 配置 Server 的地址。而 Server 节点还会参与到 WAN Gossip 池中。这样，通过 Server 节点就可以让 DataCenter 之间做简单的服务发现。比如，增加一个 Datacenter 就只需要让 Server 节点参与到 Gossip Pool 中。并且，DataCneter 之间的通信和服务请求可以通过 WAN Gossip 来随机请求另外一个 DataCenter 的 Server 节点，被请求的 Server 会再把请求 foword 到本 DataCenter 的 leader 节点。Server leader 的选举通过 Consul 的 Raft 算法实现。leader 节点需要负责所有请求和处理，并且这些请求也必须复制给所有的其他非 leader 的 Server 节点。同样，非 leader 节点接收到 RPC 请求的时候也会 foward 到 leader 节点。

任务实施

构建 Consul 集群服务

① 创建目录来保存 Consul 的 Dockerfile。

```
# mkdir consul
# cd consul/
# touch Dockerfile
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

② 编辑 Dockerfile。

```
# vi Dockerfile
FROM ubuntu:16.04
MAINTAINER James Turnbull <james@example.com>
ENV REFRESHED_AT 2016-06-01

RUN apt-get -qqy update
RUN apt-get -qqy install curl unzip

ADD https://releases.hashicorp.com/consul/0.6.4/consul_0.6.4_linux_amd64.zip /tmp/consul.zip
RUN cd /usr/sbin && unzip /tmp/consul.zip && chmod +x /usr/sbin/consul && rm /tmp/consul.zip

RUN mkdir -p /webui/
ADD https://releases.hashicorp.com/consul/0.6.4/consul_0.6.4_web_ui.zip /webui/webui.zip
RUN cd /webui && unzip webui.zip && rm webui.zip

ADD consul.json /config/

EXPOSE 53/udp 8300 8301 8301/udp 8302 8302/udp 8400 8500

VOLUME ["/data"]

ENTRYPOINT [ "/usr/sbin/consul", "agent", "-config-dir=/config" ]
```

③ 编辑 Consul.json。

```
# vi consul.json
{
  "data_dir": "/data",
  "ui_dir": "/webui",
  "client_addr": "0.0.0.0",
  "ports": {
    "dns": 53
  },
  "recursor": "8.8.8.8"
}
```

④ 构建 Consul 镜像。

```
# docker build -t="jamtur01/consul" .
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM ubuntu:16.04
Trying to pull repository 192.168.200.11:5000/ubuntu ...
Pulling repository 192.168.200.11:5000/ubuntu
Error: image ubuntu not found
Trying to pull repository docker.io/library/ubuntu ...
16.04: Pulling from docker.io/library/ubuntu
b3e1c725a85f: Already exists
4daad8bdde31: Already exists
63fe8c0068a8: Already exists
```

版权所有，严禁传播，违者自负法律责任！


```
4a70713c436f: Already exists
bd842a2105a8: Already exists
Digest: sha256:8259ab0b9a40a933cac5f06df7344a2f3ce51400b08ae38c3d0234f44d18406a
Status: Downloaded newer image for docker.io/ubuntu:16.04
 ---> 104bec311bcd
Step 2 : MAINTAINER James Turnbull <james@example.com>
 ---> Running in 00c6ed0a1cdc
 ---> 0417cfa622d9
Removing intermediate container 00c6ed0a1cdc
Step 3 : ENV REFRESHED_AT 2016-06-01
 ---> Running in 0dfcde4ea214
 ---> cdac7642c185
.........
.........
Step 13 : VOLUME /data
 ---> Running in 880bce7f192c
 ---> 5fb2be211246
Removing intermediate container 880bce7f192c
Step 14 : ENTRYPOINT /usr/sbin/consul agent -config-dir=/config
 ---> Running in 4386f5c3e2e6
 ---> 1f96baa6f731
Removing intermediate container 4386f5c3e2e6
Step 15 : CMD
 ---> Running in e64ef3e7bad1
 ---> 54c19d0ae6a9
Removing intermediate container e64ef3e7bad1
Successfully built 54c19d0ae6a9
```

⑤ 执行一个本地 Consul 节点，启动 Consul 服务。

```
# docker run -p 53:53/udp -h node1 jamtur01/consul -server -bootstrap
==> WARNING: Bootstrap mode enabled! Do not enable unless necessary
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
           Node name: 'node1'
          Datacenter: 'dc1'
              Server: true (bootstrap: true)
         Client Addr: 0.0.0.0 (HTTP: 8500, HTTPS: -1, DNS: 53, RPC: 8400)
        Cluster Addr: 172.17.0.5 (LAN: 8301, WAN: 8302)
      Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
               Atlas: <disabled>

==> Log data will now stream in as it occurs:

    2016/12/28 12:24:23 [INFO] serf: EventMemberJoin: node1 172.17.0.5
    2016/12/28 12:24:23 [INFO] serf: EventMemberJoin: node1.dc1 172.17.0.5
    2016/12/28 12:24:23 [INFO] raft: Node at 172.17.0.5:8300 [Follower] entering Follower state
    2016/12/28 12:24:23 [INFO] consul: adding LAN server node1(Addr:172.17.0.5:8300) (DC: dc1)
    2016/12/28 12:24:23 [INFO] consul: adding WAN server node1.dc1(Addr: 172.17.0.5:8300) (DC: dc1)
```

版权所有，严禁传播，违者自负法律责任！


```
    2016/12/28 12:24:23 [ERR] agent: failed to sync remote state: No cluster leader
    2016/12/28 12:24:24 [WARN] raft: Heartbeat timeout reached, starting election
    2016/12/28 12:24:24 [INFO] raft: Node at 172.17.0.5:8300 [Candidate] entering Candidate state
    2016/12/28 12:24:24 [INFO] raft: Election won. Tally: 1
    2016/12/28 12:24:24 [INFO] raft: Node at 172.17.0.5:8300 [Leader] entering Leader state
    2016/12/28 12:24:24 [INFO] consul: cluster leadership acquired
    2016/12/28 12:24:24 [INFO] consul: New leader elected: node1
    2016/12/28 12:24:24 [INFO] raft: Disabling EnableSingleNode (bootstrap)
    2016/12/28 12:24:24 [INFO] consul: member 'node1' joined, marking health alive
    2016/12/28 12:24:24 [INFO] agent: Synced service 'consul'
==> Newer Consul version available: 0.7.2
==> Caught signal: interrupt
==> Gracefully shutting down agent...
    2016/12/28 12:53:20 [INFO] consul: server starting leave
    2016/12/28 12:53:20 [INFO] serf: EventMemberLeave: node1.dc1 172.17.0.5
    2016/12/28 12:53:20 [INFO] serf: EventMemberLeave: node1 172.17.0.5
    2016/12/28 12:53:20 [INFO] agent: requesting shutdown
    2016/12/28 12:53:20 [INFO] consul: shutting down server
    2016/12/28 12:53:20 [ERR] dns: error starting tcp server: accept tcp [::]:53: use of closed network connecti
```

项目实训

【实训题目】

构建 Consul 集群服务。

【实训目的】

1. 掌握如何构建 Consul 镜像。
2. 掌握如何使用 Consul 服务构建集群。

【实训内容】

1. 创建目录来保存 Consul 的 Dockerfile。
2. 编写 Consul 的 Dockerfile 文件。
3. 编辑 Consul.json 配置文件。
4. 构建 Consul 镜像。
5. 启动 Consul 服务，构建 Consul 集群。

单元小结

经过本单元对 Docker-compose 的学习，学生对应用案例的部署有了深刻的理解，更加熟悉了应用开发的过程。通过对 Swarm、Consul 等集群服务的学习，学生更好地对容器和应用进行管理，有效地解决了容器混乱编排的问题，有效地编排能使应用得到更好的使用和管理，也能拓展学生的思维模式，提高学生的实践能力。对于应用的开发，将在下一单元继续介绍。

版权所有，严禁传播，违者自负法律责任！

单元 6 容器服务管理平台 Rancher

学习目标

知识目标

- 理解 Rancher 管理平台的安装思路。
- 了解 Rancher 管理平台的功能组件。
- 熟悉各个应用服务部署的过程。

技能目标

- 掌握 Rancher 管理平台的安装部署过程。
- 掌握 Rancher 平台的运维管理和应用服务的开发。
- 掌握持续集成环境的设计思路和实现步骤。

版权所有，严禁传播，违者自负法律责任！

笔 记

学习情境

随着 Docker 新产品的上线，公司也开发出来了各种基于 Docker 的应用服务。可是纷繁复杂的部署过程让客户使用起来很麻烦，面对这种情况，研发部给工程师小魏提出了新的要求，如何更方便快捷地部署各种应用服务？经过和团队的讨论之后，工程师小魏决定通过选择开源的容器服务管理平台来统一管理公司的所有应用服务。另外，公司在开发 Web 应用系统的过程中，每次有代码更新都需要重新编译、提交、发布，过程很是烦琐，时间和人力成本很高。面对这种情况，工程师小魏决定基于容器平台实现持续集成环境（DevOps）的部署，提高公司产品研发的效率。

任务 6.1　Rancher 安装、部署、使用

任务描述

1. 实现 Rancher 容器管理平台在 CentOS7 系统上的多节点安装、部署、使用。
2. Rancher 容器管理平台在完成安装部署之后，实现应用服务的使用和运维。
3. 整理搭建过程的操作步骤，形成规范的操作手册。

知识学习

1. Rancher 介绍

Rancher 是一个开源的项目，提供了在产品环境中对 Docker 容器进行全方位管理的平台。它提供的基础架构服务包括多主机网络、全局和局部的负载均衡、卷快照等。Rancher 集成了原生 Docker 管理能力，包括 Docker Machine 和 Docker Swarm。Rancher 还提供了丰富用户体验的管理功能。

Rancher 支持多种编排系统，支持多种 IaaS 网络服务和裸机。图 6-1-1 所示为 Rancher 编排系统。

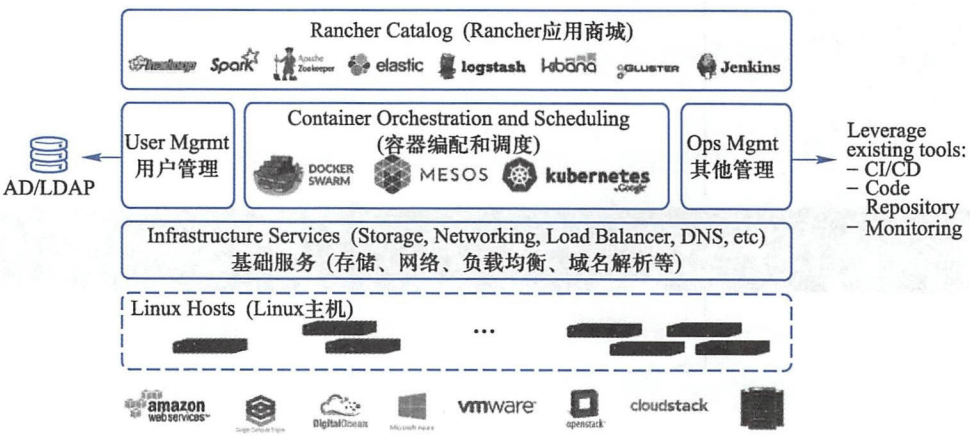


图 6-1-1
Rancher 编排系统

版权所有，严禁传播，违者自负法律责任！

2. Rancher 的商业支持

Rancher 提供的工具可以说是大而全的。在业务上，Rancher 能同时提供对 Docker Swarm、Kubernetes 和 Mesos 等多种编排调度框架的商业支持，具有满足企业多样化需求的灵活性，具体如图 6-1-2 所示。

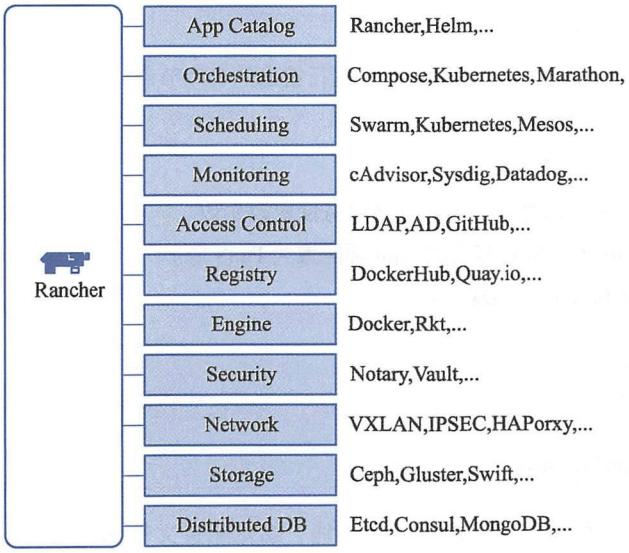


图 6-1-2
Rancher 提供的工具

任务实施

Rancher 是开源的 Docker 全栈容器服务管理平台，通过提供“应用商店”可以部署各种应用，提供 CaaS 容器服务。默认支持通过整合 Cattle、Swarm、Hubernetes、MesOS 容器等编排集群服务实现。

Rancher 提供大量的 docker hub 官方镜像，用户只要通过管理界面就可部署应用，构建集群环境。界面非常实用。

（1）准备工作

① 系统要求。

安装运行环境，系统要求为 CentOS7，内核版本不低于 3.10，Docker 版本不低于 1.10。

② 设备说明。

将结构设计为 4 个节点，也可以单节点构建，网络架构如图 6-1-3 所示。

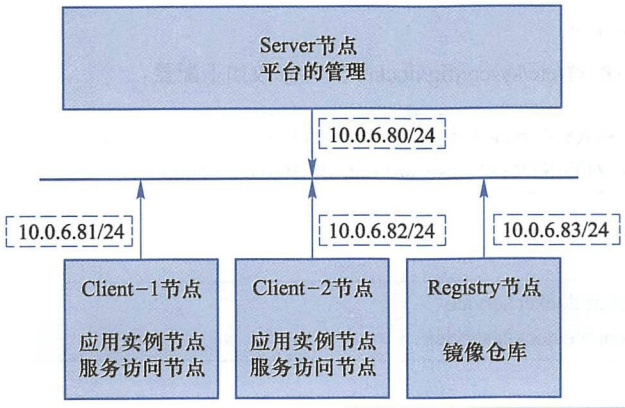


图 6-1-3
网络架构图

③ 网络说明。

- Server 节点：主要作为系统运行指令的发送节点。

server：10.0.6.80。

- Client 节点：主要为服务运行的节点，使用一个或者多个均可。

client1：10.0.6.81；

client2：10.0.6.82。

- Registry 节点：主要作为系统所有运行镜像的仓库节点。

registry：10.0.6.83。

（2）配置基础环境

在配置完网络接口之后，重启网络服务使更改生效。

① 配置 yum 源，所有节点的 yum 源配置为 PaaS 源。

② 删除 iptables 防火墙规则：

```
# iptables –F
# iptables –X
# iptables –Z
# /usr/sbin/iptables-save
```

③ 修改系统内核，打开内核转发功能，编辑配置文件/etc/sysctl.conf，添加以下内容：

```
net.ipv4.ip_forward = 1
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
```

修改完成后使用命令生效：

```
# sysctl –p
```

（3）安装服务

① 基础配置。

为所有节点安装 Docker 环境：

```
# yum -y install docker-io
```

② 配置 Docker。

为所有节点配置/etc/sysconfig/docker 文件修改如下配置：

```
ADD_REGISTRY='--add-registry 10.0.6.83:5000'
INSECURE_REGISTRY='--insecure-registry 10.0.6.83:5000'
```

③ 启动服务。

```
# systemctl start docker.service
# systemctl enable docker.service
```

④ 配置镜像仓库。

将提供的软件包复制到镜像注册节点，而后进行如下操作。

镜像仓库导入镜像：

```
# docker load < registry_latest.tar
```

创建镜像仓库：

```
#docker run -d -p 5000:5000 --restart=always --name registry docker.io/registry:latest（只需要 registry 节点执行）
```

查询本地镜像：

```
# docker image（查询上传的 image id）
```

给镜像添加标签：

```
# docker tag c9bd19d022f6（此处为上一步查看的上传镜像的 ID 值）10.0.6.83:5000/registry:latest
```

上传镜像到私有仓库：

```
# docker push 10.0.6.83:5000/registry:latest
```

以此类推，上传其他的镜像到私有仓库内。

⑤ 下载镜像。

server 节点：

```
# docker pull rancher/server:v1.1.4-xd
```

client 节点：

```
# docker pull rancher/agent-instance:v0.8.3
# docker pull rancher/agent:v1.0.2
```

⑥ 启动服务。

在 server 节点启动：

```
# docker run -d --restart=always -p 80:8080 rancher/server:v1.1.4-xd
```

运行完毕后可以通过 docker ps －a 命令查看运行的情况：

```
[root@server ~]# docker ps -a
CONTAINER ID  IMAGE                COMMAND                CREATED        STATUS          PORTS                    NAMES
1257e69def4b  rancher/             "/usr/bin/s6-          21 hours ago   Up 5            3306/tcp, 0.0.0.0:       pensive_
              server:v1.1.4-xd     svscan/"                              minutes         80->8080/tcp             tesla
```

（4）应用模板部署

① 添加主机。

主机注册 URL，图 6-1-4 所示为 Rancher 管理平台添加主机的页面。

图 6-1-4
Rancher 管理平台
添加主机的页面

Rancher 管理平台添加主机成功后的页面如图 6-1-5 所示。

图 6-1-5
Rancher 管理平台
添加主机成功后的页面

② 配置账号和密码。

在 Rancher 管理平台添加用户，界面如图 6-1-6 所示。

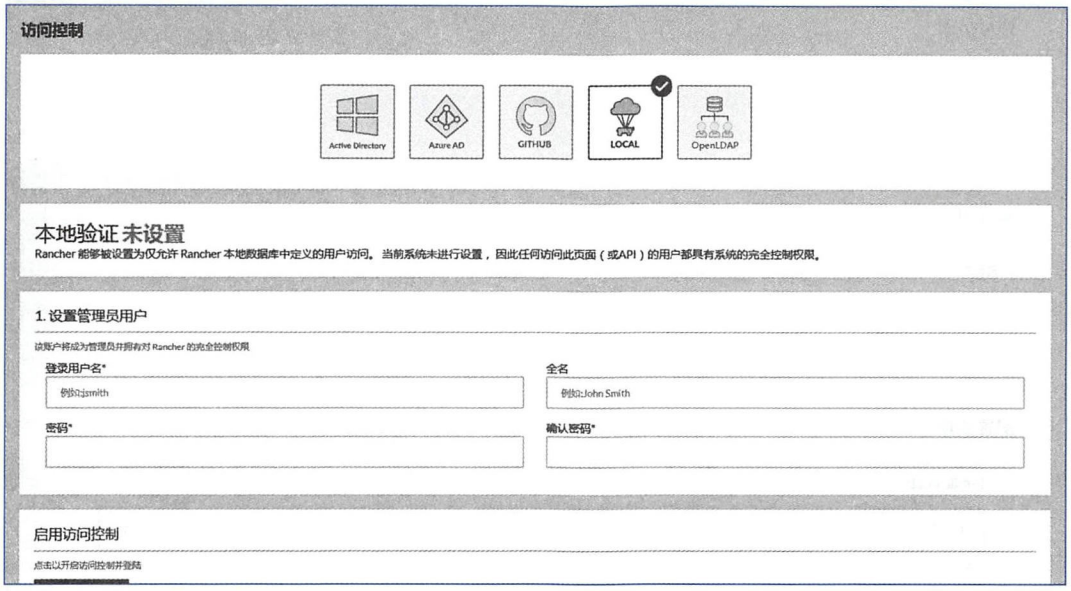


图 6-1-6 Rancher 管理平台添加用户界面

(5) 部署案例

部署博客系统，这里选用 WordPress 应用来介绍如何部署应用。

① 通过主页选择应用，Rancher 应用商店如图 6-1-7 所示。

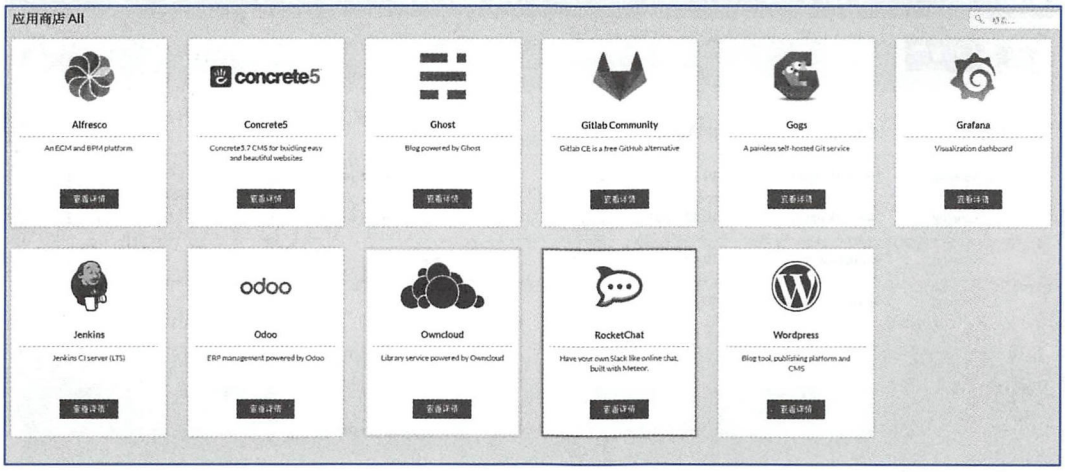


图 6-1-7 Rancher 应用商店

② 启动服务。

部署之前修改访问的端口，否则会产生端口冲突问题，WordPress 部署页面如图 6-1-8 所示。

非卖品，仅供非商业用途或交流学习使用

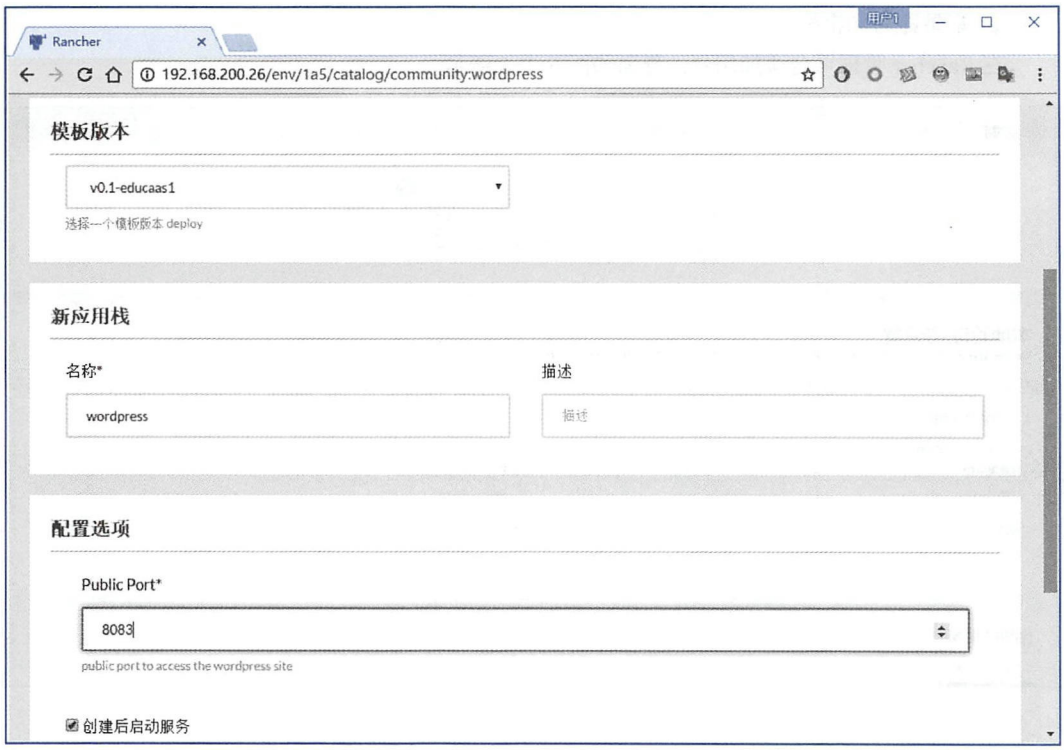


图 6-1-8 WordPress 部署页面

完成后启动，完成服务的部署，如图 6-1-9 所示。服务部署完成后，单击端口链接访问。

图 6-1-9 WordPress 部署成功

(6) 使用 Rancher 运维

① 通过创建数据容器升级 Rancher。

非卖品，仅供非商业用途或交流学习使用

停止容器：

```
$ docker stop <container_name_of_original_server>
```

② 创建一个 rancher-data 容器。

注意

如果开发者过去已经升级过并且已经有了一个 rancher-data 容器，那么此步骤可以跳过。

```
$ docker create --volumes-from <container_name_of_original_server> \
 --name rancher-data rancher/server:<tag_of_previous_rancher_server>
```

③ 拉取最近的 Rancher 服务端镜像。

注意

如果跳过此步骤并尝试运行最近的镜像，将不会自动拉取更新后的镜像。

```
$ docker pull rancher/server:latest
```

④ 启动 Rancher 服务端。

下面启动一个新的 Rancher 服务端并使用来自 rancher-data 容器的数据库。Rancher 内的任何修改都将会保存到 rancher-data 容器。如果开发者在服务端见到了关于日志锁的异常，请参考如何处理日志锁。

注意

这取决于用户运行了多久 Rancher 服务端，某些数据库迁移可能花费比预计更长的时间。请不要在升级过程中停止升级，否则在下次升级时将会遇上数据库迁移错误。

```
$ docker run -d --volumes-from rancher-data --restart=always \
  -p 8080:8080 rancher/server:latest
```

注意

如果开发者在原始的 Rancher 服务端安装时设置过任何环境参数或传入过 ldap 认证，则开发者将需要在命令中加入这些环境变量或认证。

⑤ 移除旧的 Rancher 服务端。

注意

如果开发者仅停止了容器，那么容器将会由于有--restart 而在当前机器重启后重新运行。本书建议在升级成功后移除此容器。

⑥ 通过启动时使用绑定挂载升级 Rancher。

之前介绍了通过创建数据容器升级 Rancher，这里介绍通过启动时使用绑定挂载升级 Rancher。

停止运行中的 Rancher 服务端容器：

非卖品，仅供非商业用途或交流学习使用

```
$ docker stop <container_name_of_original_server>
```

⑦ 从服务端容器中复制数据库文件。

注意

如果开发者已经有数据库存储在主机上了，那么可以跳过此步骤。另外，如果数据库已经被复制到了容器外面，那么它将会在//mysql/里面，因为 Docker 的复制方法如此。在绑定挂载到容器里时一定要考虑到这一点。如果开发者开始绑定挂载，将不需要输入 mysql/。

```
$ docker cp <container_name_of_original_server>:/var/lib/mysql <path on host>
```

⑧ 现在为文件夹设置 UID/GID 以便容器内的 mysql 用户对 mysql 挂载有正确的所有权。

```
$ sudo chown -R 102:105 <path on host>
```

⑨ 开启新的服务端容器。

```
$ docker run -d -v <path_on_host>:/var/lib/mysql -p 8080:8080 \
  --restart=always rancher/server:latest
```

注意

如果这是从前一个容器中复制出来的数据库，确保主机路径最后是“/”将很重要，否则目录会在错误的地方结束。

项目实训

【实训题目】

在 Ubuntu 系统上安装 Rancher 管理平台，并实现 Gogs 应用服务的部署。

【实训目的】

1. 掌握 Ubuntu 系统的基本命令。
2. 掌握 Rancher 管理平台在 Ubuntu 系统上的安装。
3. 掌握 Gogs 应用服务的部署和使用。

【实训内容】

1. 安装 Ubuntu 系统。
2. 安装 Rancher-Server、Rancher-Agent、Rancher-Registry 节点。
3. 部署 Rancher-Registry 节点，把 Gogs 应用服务的镜像包放到仓库内，实现自动获取。
4. 启动 Server 和 Registry 服务容器，并添加 Agent 主机到 Rancher 管理平台上。
5. 部署 Gogs 应用服务，并创建私有仓库。

任务 6.2　基于 Rancher 构建持续集成

任务描述

1. 通过安装 Jenkins 构建持续集成安排环境。

非卖品，仅供非商业用途或交流学习使用

2. 通过安装 Maven 构建持续集成自动化部署环境。
3. 通过安装 Git 和 MySQL 构建持续集成数据库环境。
4. 通过安装 Tomcat 构建持续集成发布环境。

笔记

知识学习

1. 持续集成过程

技术的发展开始呈现出高速化的姿态，在此驱动下，软件也必须实现快速化的开发以适应技术的发展。此时敏捷开发便出现了，简单地说，敏捷开发是一种以人为核心、迭代、循序渐进的开发方法。它以适应性的过程来代替传统的预测性的过程，在很大程度上满足了现代商业软件业务复杂、需求多变、时间要求紧迫等需求。

相比瀑布式开发漫长严格的开发周期，敏捷开发要求在几周或者几个月的时间内完成相对较小的功能，强调的是能尽早将尽量小的可用功能交付使用，并在整个项目周期中持续改善和增强。而相比同样强调较短开发周期的迭代式开发，敏捷开发的周期更短，并更注重队伍中的高度协作。

极限编程作为敏捷开发的方法之一，其中有一项最佳实践，即持续集成。极限编程认为：项目每天可以构建多次，每日构建是最低要求。一个完全自动化的过程应该让项目每天完成多次构建，这是可行的。在工作几个小时的开发后，就要对刚才工作的代码进行集成和测试，并快速获得反馈。持续集成虽然出自极限编程的实践，但随着持续集成的发展，它已经开始作为一种优秀的软件开发实践应用于各种开发方法中。

持续集成与“每日构建”相比有几点不同：首先，持续集成比每日构建的集成频率更高，具体依据项目而定；再者，持续集成在集成失败后向开发人员提供快速的反馈，让开发人员可以迅速修复错误，而不仅仅是在成功构建后提供一个稳定的版本；最后，持续集成鼓励开发人员频繁地提交修改后的代码并得到尽快的反馈，每次修改的代码量减少后，修复也变得容易和迅速。

2. 持续集成的原则

业界普遍认同的持续集成的原则包括以下几点：

① 需要版本控制软件保障团队成员提交的代码不会导致集成失败。常用的版本控制软件有 ClearCase、CVS、Subversion、Git 等；

② 开发人员必须及时向版本控制库中提交代码，也必须经常性地从版本控制库中更新代码到本地；

③ 需要有专门的集成服务器来执行集成构建。根据项目的具体实际，集成构建可以通过软件的修改来直接触发，也可以定时启动，如每半个小时构建一次；

④ 必须保证构建的成功。如果构建失败，则修复构建过程中的错误是优先级最高的工作。一旦修复，需要手动启动一次构建。

3. 持续集成的价值

持续集成的价值主要表现在以下方面。

① 一天中进行多次的集成，并做了相应的测试，这样有利于检查缺陷，了解软件的健康状况。

② 减少重复的过程可以节省时间、费用和工作量。说起来简单，做起来难。这些浪

非卖品，仅供非商业用途或交流学习使用

笔 记

费时间的重复劳动可能在项目活动的任何一个环节发生，包括代码编译、数据库集成、测试、审查、部署及反馈。通过自动化的持续集成可以将这些重复的动作都变成自动化的，无须太多人工干预，让人们的时间更多地投入到动脑筋的、更高价值的事情上。

③ 持续集成可以在任何时间发布可以部署的软件。从外界来看，这是持续集成最明显的好处，可以改进软件品质和减少风险，但对于客户来说，可以部署的软件产品是最实际的资产。利用持续集成，开发者可以经常对源代码进行一些小改动，并将这些改动和其他的代码进行集成。如果出现问题，项目成员马上就会被通知到，问题会第一时间被修复。在不采用持续集成的情况下，这些问题有可能在交付前的集成测试的时候才被发现，有可能会导致延迟发布产品，而在急于修复这些缺陷的时候又有可能引入新的缺陷，最终可能导致项目失败。

④ 持续集成让开发者能够注意到趋势并采取有效的决策。如果没有真实或最新的数据提供支持，则项目就会遇到麻烦。通常，项目成员手工收集这些信息，增加了负担，也很耗时。持续集成系统为项目构建状态和品质指标提供了及时的信息，有些持续集成系统可以报告功能完成度和缺陷率。再者由于经常集成，开发者可以看到一些趋势，如构建成功或失败、总体品质及其他项目信息。

⑤ 持续集成可以建立开发团队对开发产品的信心，因为他们清楚地知道每一次构建的结果，他们知道他们对软件的改动造成了哪些影响，结果怎么样。而长期稳定的成功构建将极大地鼓舞团队的士气。

4. 持续集成的应用

持续集成的应用主要有以下几方面。

① 团队开发。工程化软件开发中的开发团队规模往往上百人，且工程中又有许多分支，这时就需要统一的工程管理措施来确保准确有效的开发工作。

② 作为许多流行的软件开发理论的基础组成部分，如敏捷开发、Staging Delivery、RUP 中的迭代开发等。拿敏捷开发来说，它是一种以人为核心、迭代、循序渐进的开发方法。在敏捷开发中，软件项目的构建被切分成多个子项目，各个子项目的成果都经过测试，具备集成和可运行的特征。换言之，就是把一个大项目分为多个相互联系，但也可独立运行的小项目，并分别完成，在此过程中软件一直处于可使用状态。

③ 运营与开发。像淘宝之类的网站在进行开发、测试（压力测试、兼容性测试等）、上线（同时要装到 *N* 台服务器上）、数据维护等时均需要用到持续集成技术。

5. 持续集成的常见工具

在持续集成概念发展的十多年间，早已经出现了许多用于支持这一概念的工具，这些工具的组合使用为软件开发提供了强大的支持。

一般来说，持续集成工具可以分为两大类：自动化构建工具和构建计划安排工具。

（1）自动化构建工具

自动化构建工具有这样一些基本功能：代码编译、组件打包、程序执行和文件操作。编译源代码是构建的主要工作之一，为了提高效率，编译应该根据相应的源代码是否发生改变而有条件地执行。组件打包是将编译的结果和其他需要包含的文件组织在一起，形成可以部署的组件。构建工具应该知道何时需要重新打包。程序执行是指构建工具能够在它支持的平台上，调用所有提供命令行接口的程序。构建工具应该支持创建、复制、删除文

非卖品，仅供非商业用途或交流学习使用

件和目录等操作。

某些自动化构建工具还有一些扩展功能：执行开发者测试、版本控制工具集成、文档集成、部署功能、代码品质分析、支持扩展、多平台构建、加速构建。虽然构建工具可以通过命令行执行的方式来集成构建工具和测试工具，但如果它提供更直接的集成方式，开发者就更省力。同样，如果构建工具能够直接与版本控制工具集成，那么开发者会觉得更方便。文档集成是指构建工具能够自动从源代码中抽取并生成 API 文档。构建工具还可以将打包好的组件自动部署到目标测试环境中去。构建工具一般通过一些第三方插件对代码品质进行分析。而提供插件接口，是构建工具实现可扩展性的通用方式。如果开发的软件需要在多个平台上构建并测试，那么构建工具对多平台的支持就会带来极大的方便。对于较大的代码集，一次构建可能需要好几个小时，这为持续集成带来了一些挑战。有的构建工具支持加速构建，即在多个构建服务器的多个处理器上进行分布式构建。

常见的自动化构建工具包括 Ant、NAnt、MSBuild、make、Maven、Rake、Doxygen 等。

（2）构建计划安排工具

构建计划安排工具，通常也称为持续集成服务器，其基本功能有构建执行、版本控制集成、构建工具集成、提供反馈、为构建打上标签。构建计划安排工具的核心功能就是在特定时间执行自动化的构建，这可以通过轮询版本控制库、计划驱动或事件通知等方式来实现。大部分构建计划安排工具都支持大多数流行的版本控制系统，也支持大多数流行的构建工具。构建计划安排工具至少支持通过电子邮件提供反馈信息，有一些工具可以通过即时消息、手机短信或其他设备来提供反馈信息。大多数构建计划安排工具会提供某种类型的升序计数，作为构建版本的标签。

某些构建计划安排工具还有一些扩展功能：支持项目间的依赖关系、提供用户界面、制品发布、安全。如果项目间存在依赖关系，那么用户可能希望在被依赖的项目重新构建时，重新构建依赖于它的项目。设计良好的用户界面会在工作时为用户节约时间。制品发布是指除了得到可部署的组件之外，一些成熟的某些构建计划安排工具可以将文档、测试结果、品质分析结构和其他测量指标数据格式化，便于查看。有一些工具提供了身份认证和授权等安全方面的功能，允许用户指定谁能查看结果和修改配置。

常见的构建计划安排工具包括 AnthillPro、Continuum、CruiseControl、CruiseControl.NET、Draco.Net、Luntbuild、Jenkins 等。

任务实施

1. 安装、部署持续集成环境

① 安装 Docker 服务，配置仓库地址，并检查版本号。

```
# yum install docker-io –y
# docker -v
Docker version 1.12.6, build 1398f24/1.12.6
# echo "OPTIONS='--registry-mirror=https://mirror.ccs.tencentyun.com'" >> /etc/sysconfig/docker
# systemctl daemon-reload
# service docker restart
Redirecting to /bin/systemctl restart  docker.service
```

笔记

非卖品，仅供非商业用途或交流学习使用

② 开启 Iptables 转发功能。

```
# vi /etc/sysctl.conf
net.ipv4.ip_forward = 1
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
"sysctl.conf" 4L, 93C written
# sysctl -p
net.ipv4.ip_forward = 1
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
```

③ 拉取 Gogs 和 MySQL 的镜像到本地。

```
# docker pull docker.io/gogs/gogs:latest
Trying to pull repository docker.io/gogs/gogs ...
sha256:c04d418878636efe752ceec232ec21bb56f7ec71903d99d2435cc325fa09fccb: Pulling from docker.io/ gogs/gogs
019300c8a437: Pull complete
e9a3198556c6: Pull complete
859db010d27b: Pull complete
18f526504f1c: Pull complete
100ad7ab1dc1: Pull complete
3092c736c48f: Pull complete
Digest: sha256:c04d418878636efe752ceec232ec21bb56f7ec71903d99d2435cc325fa09fccb
Status: Downloaded newer image for docker.io/gogs/gogs:latest
# docker pull docker.io/mysql:latest
Trying to pull repository docker.io/library/mysql ...
sha256:d178dffba8d81afedc251498e227607934636e06228ac63d58b72f9e9ec271a6: Pulling from docker.io/ library/mysql
9f0706ba7422: Pull complete
2290e155d2d0: Pull complete
547981b8269f: Pull complete
2c9d42ed2f48: Pull complete
55e3122f1297: Pull complete
abc10bd84060: Pull complete
c0a5ce64f2b0: Pull complete
c4595eab8e90: Pull complete
098988cead35: Pull complete
300ca5fa5eea: Pull complete
43fdc4e3e690: Pull complete
Digest: sha256:d178dffba8d81afedc251498e227607934636e06228ac63d58b72f9e9ec271a6
Status: Downloaded newer image for docker.io/mysql:latest
# docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
docker.io/gogs/gogs  latest      ff85ed8f5167      36 hours ago     127.5 MB
docker.io/mysql      latest      44a8e1a5c0b2      6 days ago       407 MB
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

④ 启动 Gogs 容器和 MySQL 容器。

```
# docker run -d -p 90:3000 --name gogs     docker.io/gogs/gogs:latest
3842ffa7db240c677e8440f9db2446308230245d8d671309795c3000679a0b5d
[root@VM_196_198_CentOS ~]# docker ps -a
CONTAINER ID  IMAGE          COMMAND          CREATED        STATUS        PORTS              NAMES
3842ffa7db24  docker.io/gogs/  "/app/gogs/      6 seconds      Up 4          22/tcp, 0.0.0.0:   gogs
              gogs:latest      docker/sta"      ago            seconds       90->3000/tcp
# docker run -d -p 13306:3306 -e MYSQL_ROOT_PASSWORD=000000 --name gogs-mysql docker.io/mysql:latest
25d354ad8947661043fc5a9c54520a5dfc5c5c197a827f626ab7af838b70ff61
# docker ps -a
CONTAINER ID  IMAGE          COMMAND          CREATED        STATUS        PORTS              NAMES
25d354ad8947  docker.io/       "docker-         7 seconds      Up 5          0.0.0.0:13306      gogs-
              mysql:latest     entrypoint.sh"   ago            seconds       ->3306/tcp         mysql
3842ffa7db24  docker.io/gogs/  "/app/gogs/      About a        Up About      22/tcp, 0.0.0.0:   gogs
              gogs:latest      docker/sta"      minute ago     a minute      90->3000/tcp
```

⑤ 进入 MySQL 容器，创建 Gogs 数据库。

```
# docker run -d -p 13306:3306 -e MYSQL_ROOT_PASSWORD=000000 --name gogs-mysql docker.io/mysql:latest
25d354ad8947661043fc5a9c54520a5dfc5c5c197a827f626ab7af838b70ff61
# docker ps -a
CONTAINER ID  IMAGE          COMMAND          CREATED        STATUS        PORTS              NAMES
25d354ad8947  docker.io/       "docker-         7 seconds      Up 5          0.0.0.0:13306      gogs-
              mysql:latest     entrypoint.sh"   ago            seconds       ->3306/tcp         mysql
3842ffa7db24  docker.io/gogs/  "/app/gogs/      About a        Up About      22/tcp, 0.0.0.0:   gogs
              gogs:latest      docker/sta"      minute ago     a minute      90->3000/tcp
# docker exec -it 25d354ad8947 bash
root@25d354ad8947:/# mysql -uroot -p000000
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.18 MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
| sys                |
+--------------------+
4 rows in set (0.01 sec)

mysql> create database gogs;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| gogs               |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql> exit
Bye
root@25d354ad8947:/# exit
exit
```

⑥ 在网页上登录 Gogs 服务，设置数据库连接地址、用户、密码，如图 6-2-1 所示。

图 6-2-1
Gogs 数据库设置

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

⑦ 进行 Gogs 服务用户注册、登录，注册如图 6-2-2 所示。

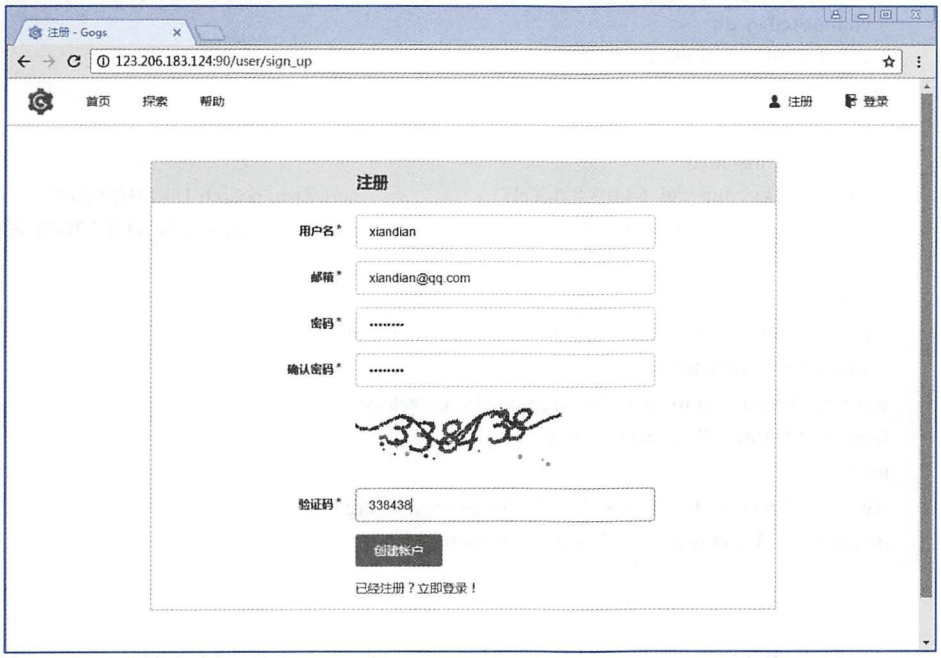


图 6-2-2
Gogs 服务注册

⑧ 在 Gogs 服务中创建新仓库，如图 6-2-3 所示。

图 6-2-3
Gogs 创建新仓库

版权所有，严禁传播，违者自负法律责任！

⑨ 复制 Git 库到本地，构建私有仓库。

```
# yum install -y git
Loaded plugins: fastestmirror, langpacks
epel

Dependency Installed:
  libgnome-keyring.x86_64 0:3.8.0-3.el7          perl-Error.noarch 1:0.17020-2.el7
  perl-Git.noarch 0:1.8.3.1-6.el7_2.1            perl-TermReadKey.x86_64 0:2.30-20.el7

Complete!
# git clone http://123.206.183.124:90/xiandian/xiandian.git
Cloning into 'xiandian'...
warning: You appear to have cloned an empty repository.
[root@VM_196_198_CentOS ~]# ll
total 8
-rw-------. 1 root root 2523 Apr 21  2016 anaconda-ks.cfg
drwxr-xr-x  3 root root 4096 Jun 30 11:07 xiandian
```

2. 安装部署 Jenkins 服务

① 利用 WinSCP 工具进行文件传输，把 apache-maven-3.5.0-bin.tar.gz、apache-tomcat-7.0.56.zip、jdk-8u77-linux-x64.tar.gz 文件复制到“/opt”目录下，如图 6-2-4 和图 6-2-5 所示。

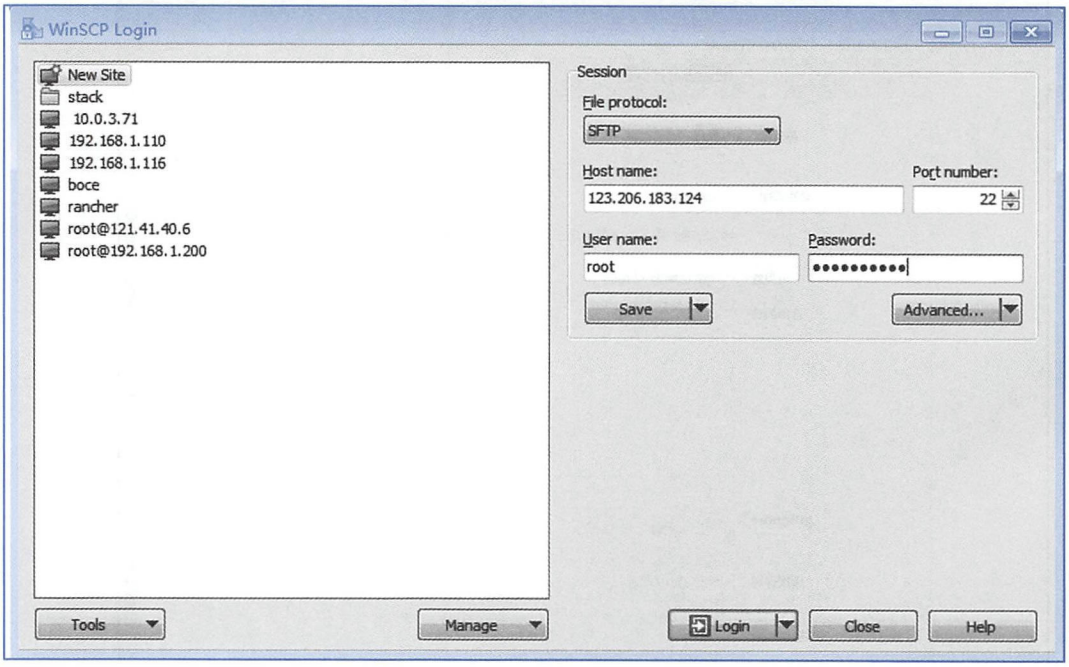


图 6-2-4 配置 WinSCP 连接

版权所有，严禁传播，违者自负法律责任！

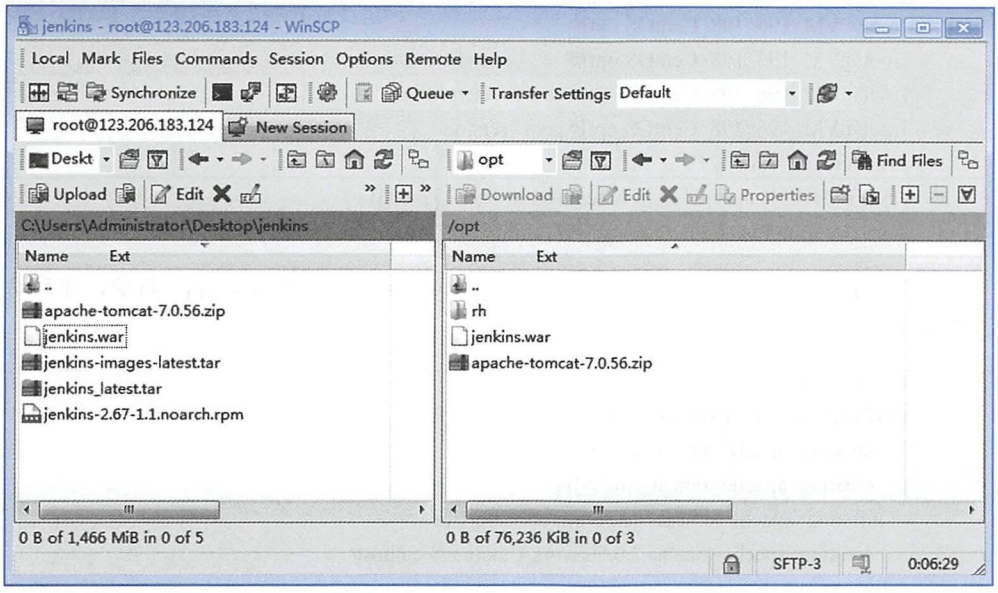


图 6-2-5 WinSCP 传输

```
# ls -l
total 261976
-rw-r--r--  1 root root    8534562 Jun 24 14:27 apache-maven-3.5.0-bin.tar.gz
-rw-r--r--  1 root root    7453955 Mar 22 08:44 apache-tomcat-7.0.56.zip
-rw-r--r--  1 root root  181365687 Jan  6 09:15 jdk-8u77-linux-x64.tar.gz
-rw-r--r--  1 root root   70611723 Jun 27 16:31 jenkins.war
drwxr-xr-x. 2 root root       4096 Mar 26  2015 rh
```

② 安装 JDK 和 Maven，生成编译环境。

```
# mkdir /usr/jdk64/
# tar -zxvf jdk-8u77-linux-x64.tar.gz -C /usr/jdk64/
jdk1.8.0_77/jre/THIRDPARTYLICENSEREADME-JAVAFX.txt
jdk1.8.0_77/jre/Welcome.html
jdk1.8.0_77/jre/README
jdk1.8.0_77/README.html
# tar -zxvf apache-maven-3.5.0-bin.tar.gz -C /usr/local/
apache-maven-3.5.0/README.txt
apache-maven-3.5.0/LICENSE
apache-maven-3.5.0/NOTICE
apache-maven-3.5.0/lib/
apache-maven-3.5.0/lib/maven-resolver-transport-wagon-1.0.3.jar
apache-maven-3.5.0/lib/maven-slf4j-provider-3.5.0.jar
apache-maven-3.5.0/lib/jansi-1.13.jar
# vi /etc/profile
# /etc/profile
export JAVA_HOME=/usr/jdk64/jdk1.8.0_77
export PATH=$JAVA_HOME/bin:$PATH
if [ -x /usr/bin/id ]; then
"/etc/profile" 79L, 1855C written
[root@VM_196_198_CentOS opt]#
```

版权所有，严禁传播，违者自负法律责任！


```
[root@VM_196_198_CentOS opt]#
[root@VM_196_198_CentOS opt]#
[root@VM_196_198_CentOS opt]# source /etc/profile
[root@VM_196_198_CentOS opt]# java -version
java version "1.8.0_77"
Java(TM) SE Runtime Environment (build 1.8.0_77-b03)
Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03, mixed mode)
```

③ 解压 Tomcat，把 jenkins.war 包放到 Tomcat 服务器的 webapps 目录，并启动 Tomcat 服务器。

```
# unzip apache-tomcat-7.0.56.zip
Archive:  apache-tomcat-7.0.56.zip
   creating: apache-tomcat-7.0.56/
   creating: apache-tomcat-7.0.56/bin/
  inflating: apache-tomcat-7.0.56/bin/bootstrap.jar
   creating: apache-tomcat-7.0.56/work/Catalina/localhost/
# ll
total 261980
-rw-r--r--  1 root root     8534562 Jun 24 14:27 apache-maven-3.5.0-bin.tar.gz
drwxr-xr-x  9 root root        4096 Oct 22  2014 apache-tomcat-7.0.56
-rw-r--r--  1 root root     7453955 Mar 22 08:44 apache-tomcat-7.0.56.zip
-rw-r--r--  1 root root 181365687 Jan  6 09:15 jdk-8u77-linux-x64.tar.gz
-rw-r--r--  1 root root    70611723 Jun 27 16:31 jenkins.war
drwxr-xr-x. 2 root root        4096 Mar 26  2015 rh

# cp jenkins.war /opt/apache-tomcat-7.0.56/webapps/
# cd /opt/apache-tomcat-7.0.56/webapps/
# ll
total 69032
-rw-r--r-- 1 root root 70611723 Jun 30 10:40 jenkins.war
# cd /opt/apache-tomcat-7.0.56/bin
# chmod +x *.sh
# ./startup.sh
Using CATALINA_BASE:    /opt/apache-tomcat-7.0.56
Using CATALINA_HOME:    /opt/apache-tomcat-7.0.56
Using CATALINA_TMPDIR: /opt/apache-tomcat-7.0.56/temp
Using JRE_HOME:         /usr/jdk64/jdk1.8.0_77
Using CLASSPATH:/opt/apache-tomcat-7.0.56/bin/bootstrap.jar:/opt/apache-tomcat-7.0.56/
bin/tomcat -juli.jar
Tomcat started.
# tail -f ../logs/catalina.out
Jun 30, 2017 10:42:09 AM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Jun 30, 2017 10:42:09 AM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
Jun 30, 2017 10:42:10 AM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
Jun 30, 2017 10:42:12 AM hudson.model.DownloadService$Downloadable load
```

版权所有，严禁传播，违者自负法律责任！


```
INFO: Obtained the updated data file for hudson.tools.JDKInstaller
Jun 30, 2017 10:42:12 AM hudson.model.AsyncPeriodicWork$1 run
INFO: Finished Download metadata. 10,377 ms
```

④ 进入网页查看 Jenkins 服务状态，并进行配置，如图 6-2-6 所示。

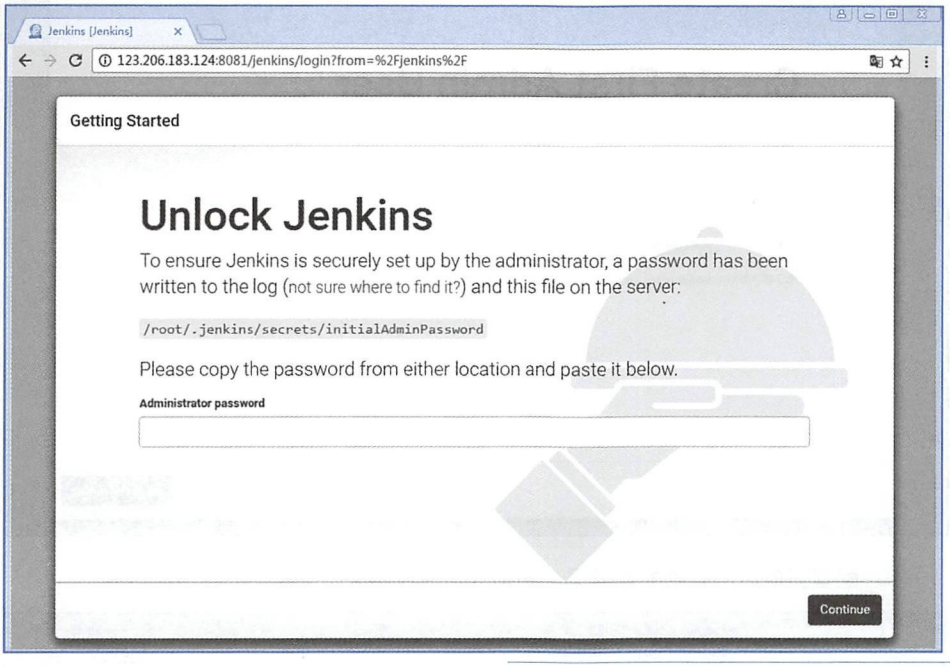


图 6-2-6
Jenkins 服务状态

⑤ 进入主机 CLI，根据网页提示地址，找到密码并输入到 Jenkins 服务中。

```
# cat /root/.jenkins/secrets/initialAdminPassword
b34a6d7cfb0445a3a4587485ff86b280
```

⑥ 在 Jenkins 网页上安装服务所需要的插件，如图 6-2-7 所示。

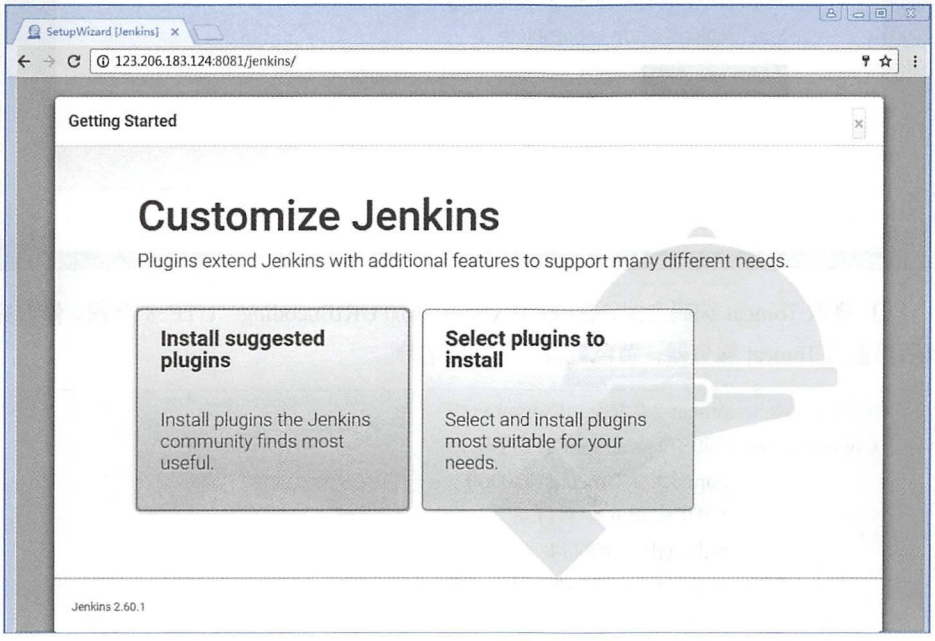


图 6-2-7
Jenkins 服务安装

版权所有，严禁传播，违者自负法律责任！

⑦ 设置登录用户和密码，如图 6-2-8 所示。

图 6-2-8
设置登录用户和密码

⑧ 配置完成后，查看配置结果，如图 6-2-9 所示。

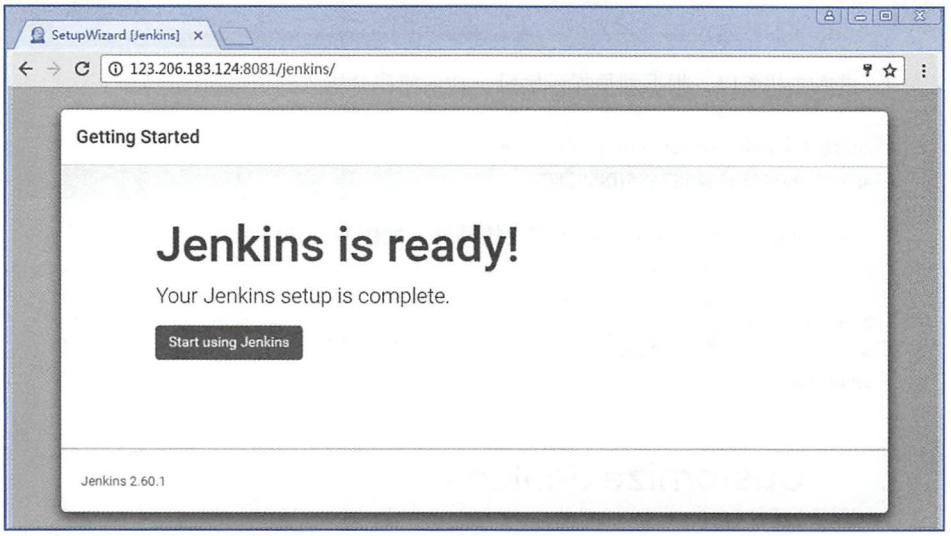


图 6-2-9
查看配置结果

⑨ 修改 Tomcat 编码方式，在 server.xml 中添加 URIEncoding="UTF-8"字段，修改完以后要重启 Tomcat 服务器，黄色警告信息就会消除。

```
# vi /opt/apache-tomcat-7.0.56/conf/server.xml
<Connector port="8081" protocol="HTTP/1.1"
                connectionTimeout="20000"
                URIEncoding="UTF-8"
                redirectPort="8444" />
    <!-- A "Connector" using the shared thread pool-->
"server.xml" [dos] 144L, 6715C written
```

版权所有，严禁传播，违者自负法律责任！

⑩ 配置 Tomcat 用户名和密码。

```
# vi /opt/apache-tomcat-7.0.56/conf/tomcat-users.xml
<tomcat-users>
<!--
  NOTE:  By default, no user is included in the "manager-gui" role required
to operate the "/manager/html" web application.  If you wish to use this app,
you must define such a user - the username and password are arbitrary.
-->
<!--
  NOTE:  The sample user and role entries below are wrapped in a comment
and thus are ignored when reading this file. Do not forget to remove
<!.. ..>that surrounds them.
-->
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
<role rolename="manager-jmx"/>
<role rolename="manager-status"/>
<user username="xiandian" password="xiandian"roles="manager-gui,manager-script,manager-
jmx, manager- status"/>

</tomcat-users>
"tomcat-users.xml" [dos] 35L, 1567C written
```

⑪ 进行 JDK 的配置，单击“系统管理”按钮，选择“Global Tool Configuration”命令，单击“新增 JDK”按钮，别名可以随便取；取消选择“自动安装”复选框；可以使用服务器本机的 JDK，在“JAVA_HOME”文本框中输入绝对路径就可以了，如图 6-2-10 所示。如果路径不对或者找不到 JDK，Jenkins 会自动提示。

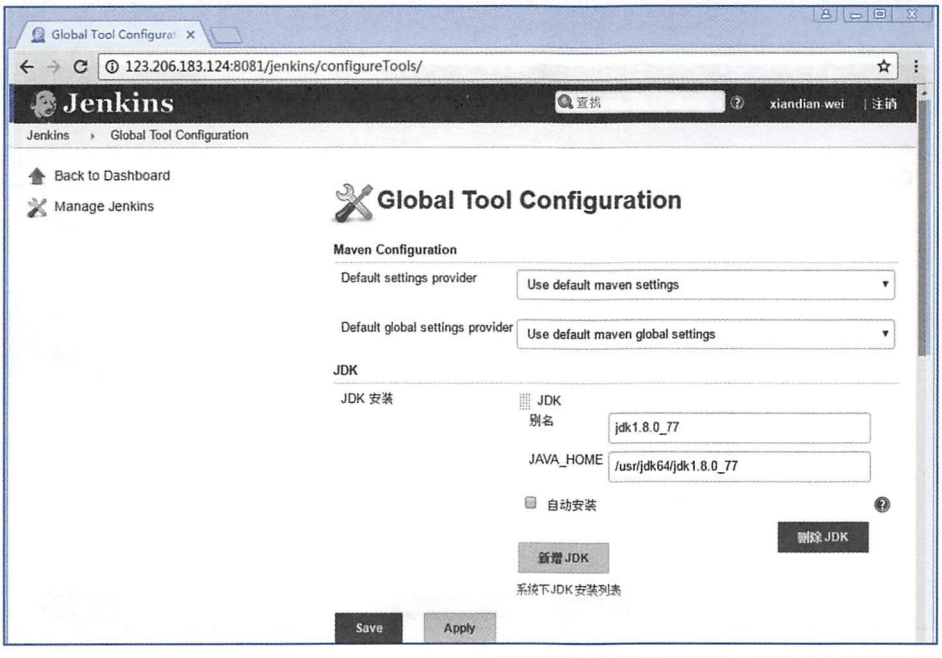


图 6-2-10
JDK 配置

版权所有，严禁传播，违者自负法律责任！

⑫ 进行 Maven 的配置，单击“系统管理”按钮，选择“Global Tool Configuration”命令，单击“新增 Maven”按钮，别名可以随便取；取消选择“自动安装”复选框；在“MAVEN_HOME”文本框中输入绝对路径就可以了，如图 6-2-11 所示。如果路径不对或者找不到 Maven，Jenkins 会自动提示。

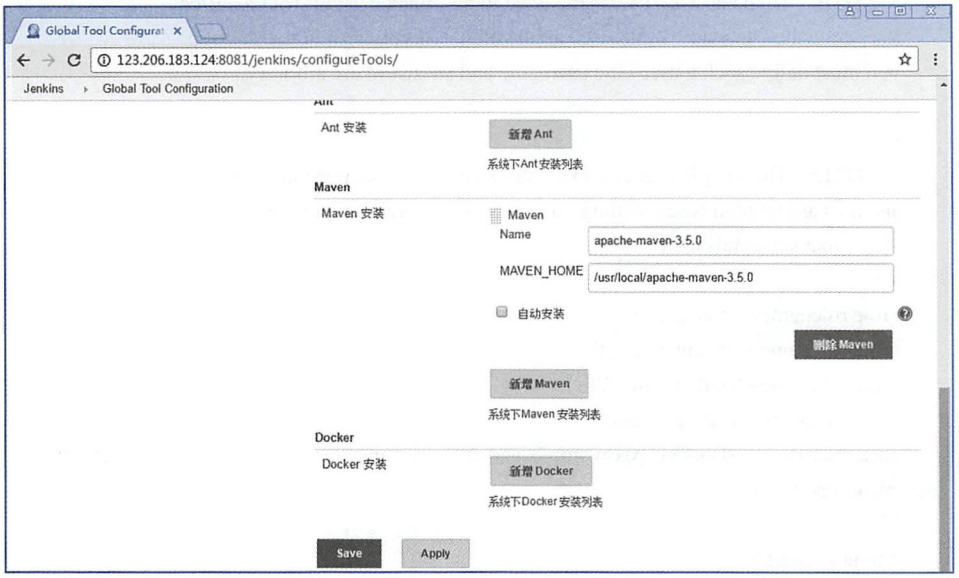


图 6-2-11
Maven 配置

⑬ 在进行新建项目之前还有一些步骤需要操作，不然仍无法新建 maven 项目。单击“系统管理”按钮，选择“插件管理”命令，单击“可选插件”标签，在“过滤”文本框中输入“maven”，选中“Maven Integration plugin”复选框，单击“直接安装”按钮，如图 6-2-12 所示。

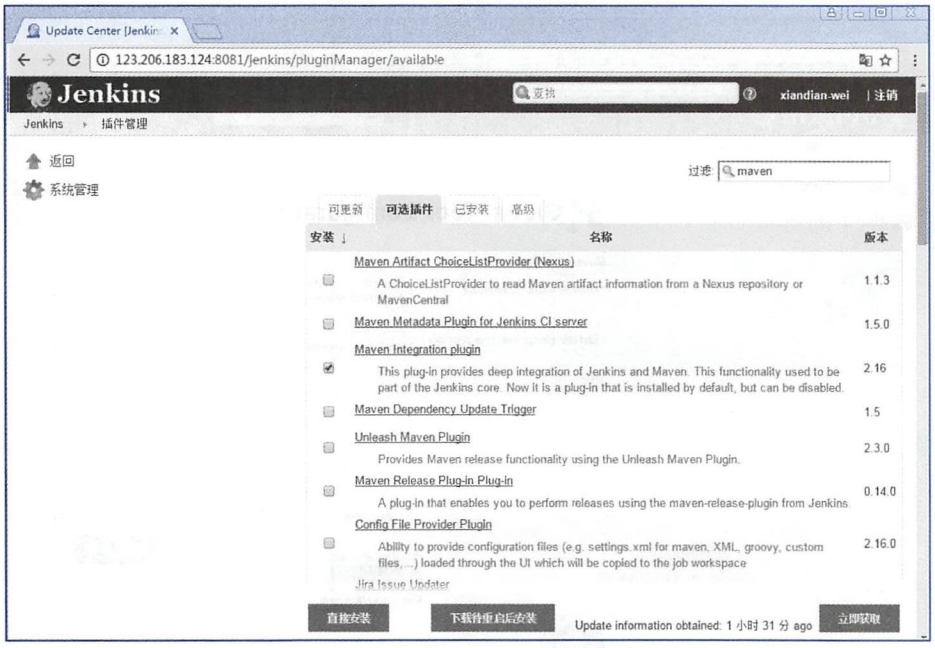


图 6-2-12
环境配置

⑭ 在使用 Jenkins 部署项目的时候，还需要一个插件，单击“系统管理”按钮，选择“插件管理”命令，同理，在“可选插件”选项卡下，在“过滤”文件框中输入“deploy”，选择“Deploy to container Plugin”复选框，单击“直接安装”按钮，等待安装完成，如图 6-2-13 所示。

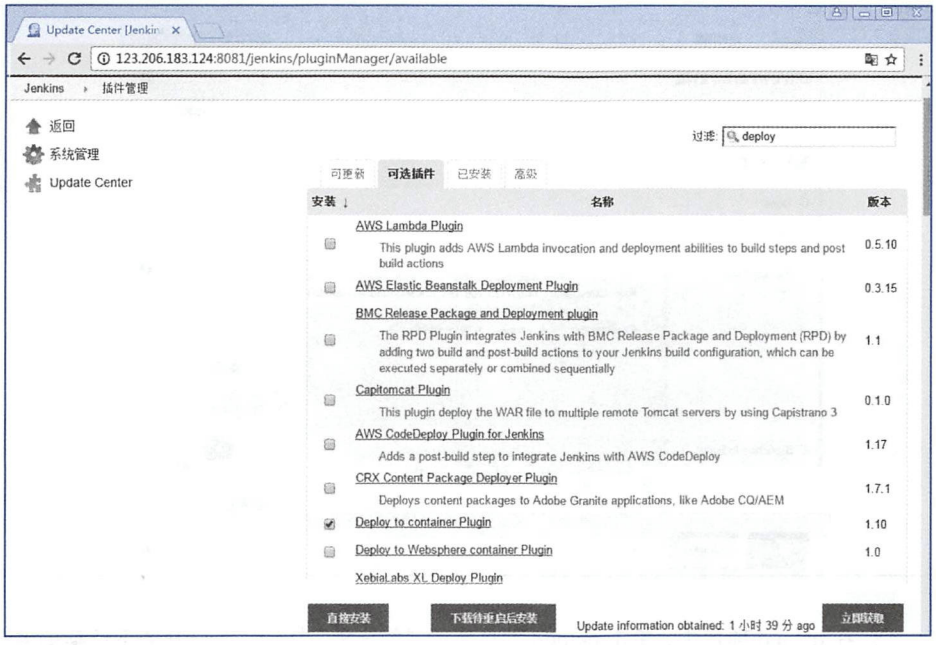


图 6-2-13
Jenkins 项目部署

3. 创建 maven 项目，并构建、发布

① 进入 Jenkins 主页，创建一个新任务，选择 maven 项目，如图 6-2-14 所示。

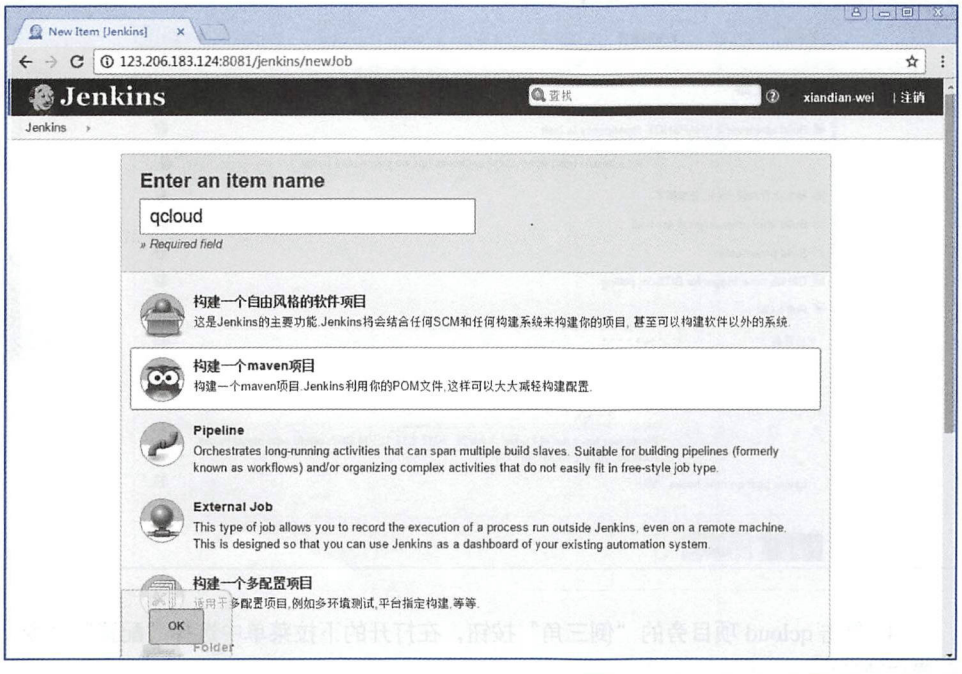


图 6-2-14
新建 maven 项目

② 在 qcloud 项目中，单击 qcloud 项目旁的“倒三角”按钮，在打开的下拉菜单中选择“配置”命令，填写正确的 Git 源，如图 6-2-15 所示。

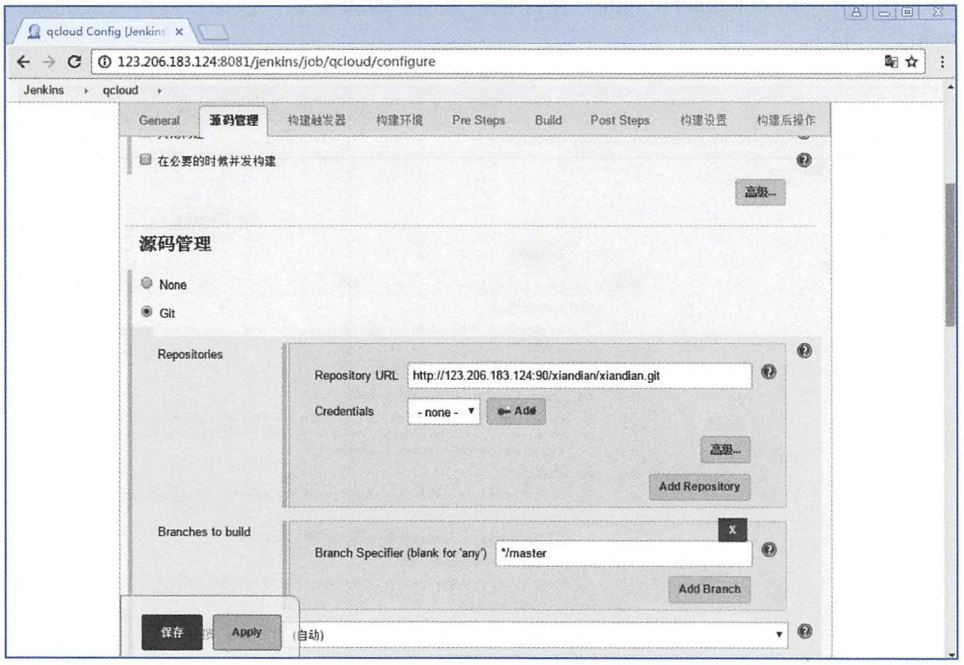


图 6-2-15
Git 源配置

③ 在 qcloud 项目中，单击 qcloud 项目旁的“倒三角”按钮，在打开的下拉菜单中选择“配置”命令，配置构建触发器，如图 6-2-16 所示。

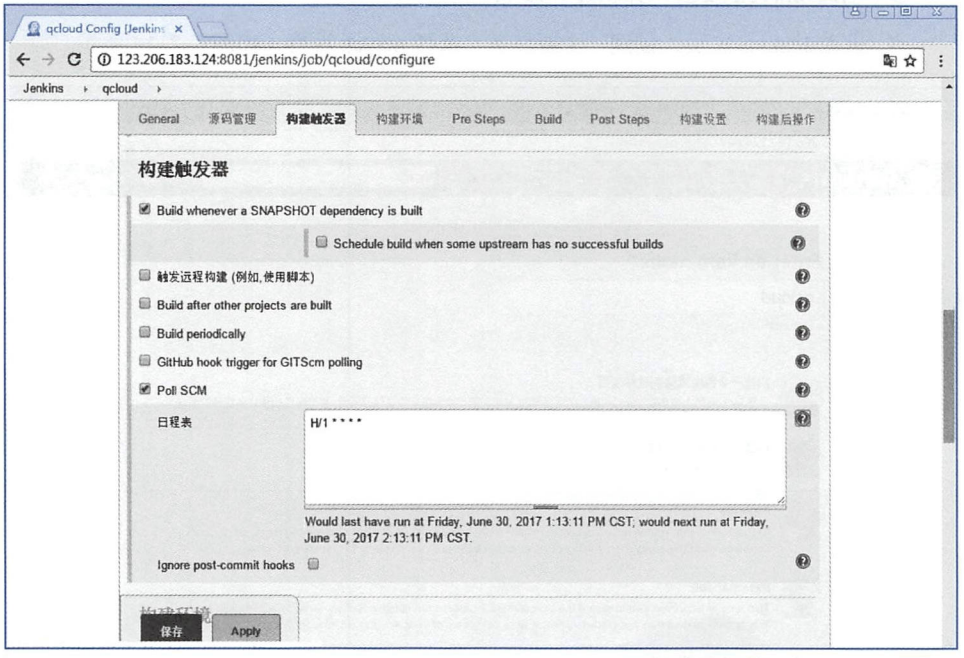


图 6-2-16
配置构建触发器

④ 单击 qcloud 项目旁的“倒三角”按钮，在打开的下拉菜单中选择“配置”命令，配置构建环境，如图 6-2-17 所示。

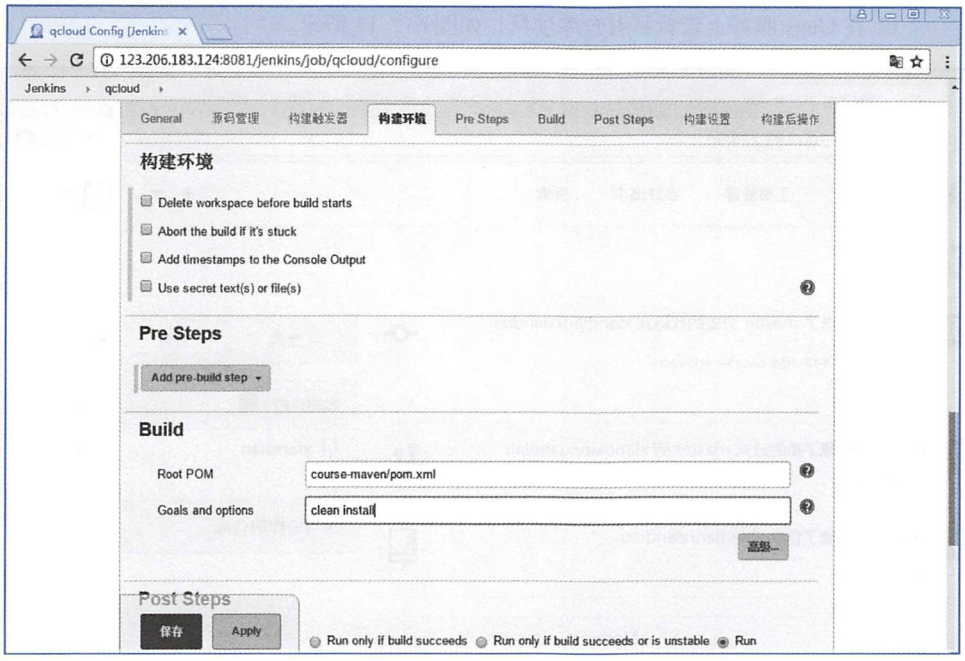


图 6-2-17
配置构建环境

⑤ 配置 Maven 本地仓库，添加 Maven 本地仓库地址，并把 maven 项目推到私有仓库。

```
# vi /usr/local/apache-maven-3.5.0/conf/settings.xml
<?xml version="1.0" encoding="UTF-8"?>
<localRepository>/opt/maven-repository</localRepository>

# git add course-maven
# git config --global user.email "xiandian@example.com"
# git config --global user.name "cetc55"
# git commit -am "add course-maven "
[master (root-commit) 3f4d687] add index.html
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
# git remote add origin http://111.231.102.198:90/cetc55/xiandian.git
fatal: remote origin already exists.
# git push -u origin master
Counting objects: 1394, done.
Compressing objects: 100% (1287/1287), done.
xiandian for 'http://123.206.183.124:90': Writing objects:   20% (280/1394), 1.10 MiB | 1.09 MiB/s
Password for 'http://xiandian@123.206.183.124:90':
Writing objects: 100% (1394/1394), 11.05 MiB | 116.00 KiB/s, done.
Total 1394 (delta 134), reused 0 (delta 0)
remote: Resolving deltas: 100% (134/134), done.
To http://123.206.183.124:90/xiandian/xiandian.git
 * [new branch]          master -> master
Branch master set up to track remote branch master from origin.
```

⑥ 在 Gogs 服务上查看私有仓库项目，如图 6-2-18 所示。

图 6-2-18 查看私有仓库项目

⑦ 修改数据库连接地址、用户、密码，如图 6-2-19 所示。

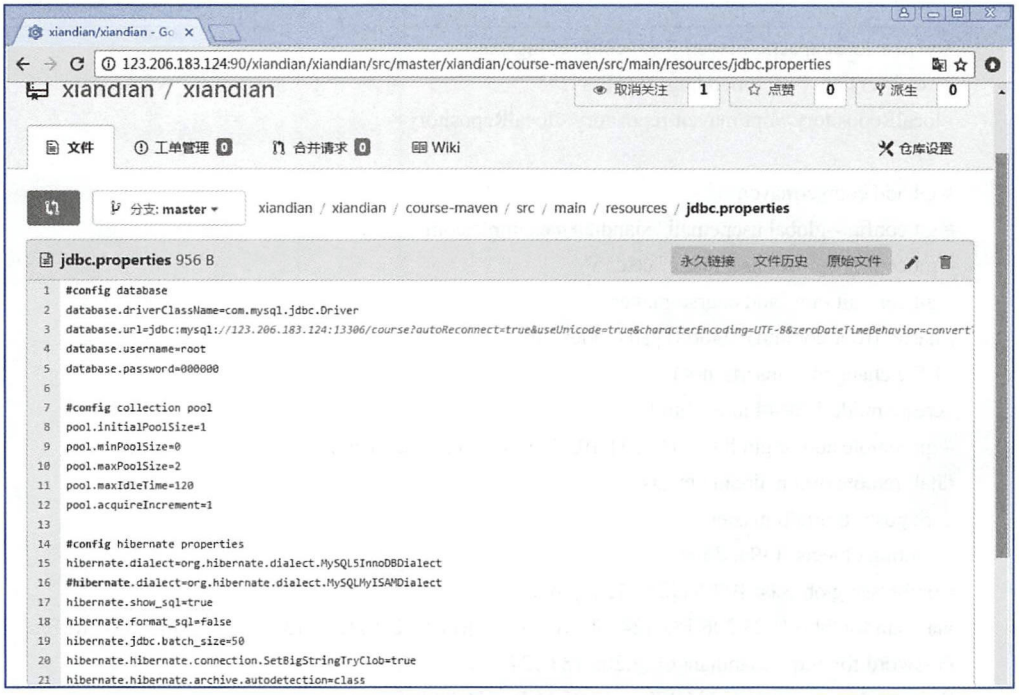


图 6-2-19 修改数据库参数

⑧ 进入 qcloud 项目，单击“立即构建”按钮，完成构建 maven 项目，并通过 Console Output 输出 log，如图 6-2-20 所示。

非卖品，仅供非商业用途或交流学习使用

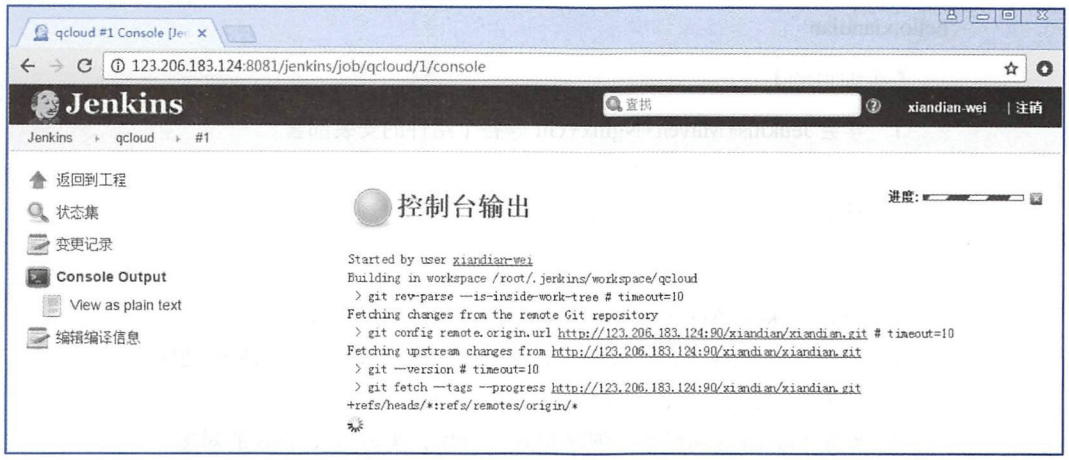


图 6-2-20 构建 maven 项目

⑨ 构建成功后查看生成的项目 WAR 包，并复制到 Tomcat 上进行发布。

```
# cd /root/.jenkins/workspace/qcloud/xiandian/course-maven/target
# cp ROOT.war /opt/apache-tomcat-7.0.56/webapps/
```

⑩ 通过网页查看项目发布的结果，如图 6-2-21 所示。

图 6-2-21 查看项目发布的结果

项目实训

【实训题目】

通过 Jenkins+Maven+Nginx+Git 实现持续集成环境的部署，实现自动编译，网页输出

非卖品，仅供非商业用途或交流学习使用

“hello,xiandian!”。

【实训目的】

1. 学会 Jenkins+Maven+Nginx+Git 等各个组件的安装部署。
2. 掌握 Jenkins+Maven+Nginx+Git 等组件之间的调用关系。
3. 掌握持续集成的设计思路和实现方法。

【实训内容】

1. 下载 Jenkins+Maven+Nginx+Git 等各个组件的镜像文件。
2. 搭建 Docker 环境，部署 Jenkins 应用服务，并安装项目需要的插件。
3. 部署 Git 应用服务，创建私有仓库与 Jenkins 进行对接。
4. 部署 Maven 应用服务，创建 Maven 仓库，实现与 Jenkins 的对接。
5. 部署 Nginxweb 服务器，实现与 Jenkins 的对接。
6. 在 Jenkins 应用服务上创建项目，编写构建命令，实现自动调用 Nginx 服务器来发布“hello,xiandian!”。
7. 在 Jenkins 应用服务上构建项目，查看控制台输出信息，进行正常调试。
8. 调试编译通过后，查看网页实现效果。

单元小结

本单元通过对 Rancher 管理平台的学习，学生了解了服务自动化部署的过程，明白了管理平台存在的重要性。通过对 Rancher 管理平台的运维，拓展了学生对 Docker 学习的广度，并加大了学生对 Docker 学习的深度。通过对 Jenkins+Maven+Nginx+Git 等各个组件的学习，学生熟悉了持续集成环境设计的思路和实现步骤。通过本单元的学习，相信学生对 Docker 容器技术有了更深的认识，能力也应该有了进一步的提升。

非卖品，仅供非商业用途或交流学习使用

单元 7 使用 Docker 构建持续集成

学习目标

知识目标

- 了解 Dockerfile 的编写方式。
- 理解通过 Dockerfile 构建应用的过程。
- 理解弹性扩展的设计思路。
- 熟悉弹性扩展的实现方法。

技能目标

- 掌握 Dockerfile 构建应用的语法规则。
- 掌握在容器中安装各种应用和软件包。
- 理解负载均衡的设计原理，掌握负载均衡的实现过程。

非卖品，仅供非商业用途或交流学习使用

笔记

学习情境

随着开发进度的推进，公司的应用产品逐渐饱和，过多地依赖管理平台，局限了新产品的开发。对此，研发部提出必须要开发出新的构建应用服务。面对研发部提出的要求，工程师小魏考虑通过使用 Dockerfile 来手动构建应用。随着使用人数的递增和访问量的倍增，应用服务的稳定性遭受了很大的冲击，为了保证服务的正常运行，研发部提出通过负载均衡来给应用服务减压分流。对于研发部提出的要求，小魏带领团队仔细商量，研究各种实现方法，最终决定通过扩展应用服务实现负载均衡。

任务 7.1　构建 Nginx Web 应用

任务描述

1. 将 Nginx Web 服务器安装到容器上，构建一个简单的网站。
2. 为 Web 应用编写一个 Dockerfile，构建一个新的 Nginx 镜像。
3. 修改网站实例，查看网页信息，验证发布结果。

知识学习

1. Dockerfile 介绍

Dockerfile 是一种被 Docker 程序解释的脚本，Dockerfile 由一条一条的指令组成，每条指令对应 Linux 下面的一条命令。Docker 程序将这些 Dockerfile 指令翻译成真正的 Linux 命令。Dockerfile 有自己的书写格式和支持的命令，Docker 程序解决这些命令间的依赖关系，类似于 Makefile。Docker 程序将读取 Dockerfile，根据指令生成定制的 image。相比 image 这种黑盒子，Dockerfile 这种显而易见的脚本更容易被使用者接受，它明确地表明 image 是怎么产生的。有了 Dockerfile，当需要定制自己额外的需求时，只需在 Dockerfile 上添加或者修改指令，重新生成 image 即可，省去了敲命令的麻烦。

Dockerfile 指令的使用方法如下：

FROM：指定基础 image。

MAINTAINER：用来指定镜像创建者信息。

RUN：用来进行软件安装。

CMD：设置 container 启动时执行的操作。

注意

该指令只能在文件中存在一次，如果有多个，则只执行最后一条。

ENTRYPOINT：设置 container 启动时执行的操作。

注意

可以多次设置，但是只有最后一个有效。

非卖品，仅供非商业用途或交流学习使用

EXPOSE：指定容器需要映射到宿主机器的端口。

ENV：用于设置环境变量。

ADD：从 src 复制文件到 container 的 dest 路径。

VOLUME：指定挂载点。

WORKDIR：切换目录，相当于 cd。

ONBUILD：在子镜像中执行。

2. Nginx 服务器介绍

Nginx（engine x）是一个高性能的 HTTP 和反向代理服务器，也是一个 IMAP/POP3/SMTP 服务器。Nginx 是由伊戈尔·赛索耶夫为俄罗斯访问量第二的 Rambler.ru 站点（俄文：Рамблер）开发的，第一个公开版本 0.1.0 发布于 2004 年 10 月 4 日。其将源代码以类 BSD 许可证的形式发布，因它的稳定性好、丰富的功能集、示例配置文件和低系统资源的消耗而闻名。2011 年 6 月 1 日，Nginx 1.0.4 发布。

Nginx 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器，并在一个 BSD-like 协议下发行。其特点是占用内存少，并发能力强，事实上，Nginx 的并发能力确实在同类型的网页服务器中表现较好。中国大陆使用 Nginx 网站用户有百度、京东、新浪、网易、腾讯、淘宝等。

任务实施

① 创建 Dockerfile 文件。

首先创建一个目录 sample，保存 Dockerfile。

```
# mkdir sample
# cd sample
# touch Dockerfile
```

② 配置 Nginx 文件。

创建一个名为 nginx 的目录，用来存放这些配置文件，然后从 GitHub 上下载实例文件。

```
# mkdir nginx && cd nginx
# wget https://raw.githubusercontent.com/jamtur01/dockerbook-code/master/code/5/sample/nginx/global.conf
```

实例文件下载成功后，展示一下 global.conf，内容如下：

```
server {
        listen          0.0.0.0:80;
        server_name     _;
        root            /var/www/html/website;
        index           index.html index.htm;
        access_log      /var/log/nginx/default_access.log;
        error_log       /var/log/nginx/default_error.log;
}
[root@l-client nginx]#wget https://raw.githubusercontent.com/jamtur01/dockerbook-code/master/
code/5/ sample/nginx/nginx.conf
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

下载成功后，展示一下 nginx.conf，内容如下：

```
user www-data;
worker_processes 4;
pid /run/nginx.pid;
daemon off;

events { }

http {
  sendfile on;
  tcp_nopush on;
  tcp_nodelay on;
  keepalive_timeout 65;
  types_hash_max_size 2048;
  include /etc/nginx/mime.types;
  default_type application/octet-stream;
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;
  gzip on;
  gzip_disable "msie6";
  include /etc/nginx/conf.d/*.conf;
}
```

③ 为示例网站编写 Dockerfile。

```
# vi Dockerfile
```

Dockerfile 内容如下：

```
FROM ubuntu:14.04.3
MAINTAINER James Turnbull "james@example.com"
NV REFRESHED_AT 2016-12-25
RUN sudo apt-get -yqq install nginx
RUN mkdir -p /var/www/html/website
ADD nginx/global.conf /etc/nginx/conf.d/
ADD nginx/nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
```

这个简单的 Dockerfile 内容包括以下几项。

- 安装 Nginx。
- 在容器中创建一个目录/var/www/html/website/。
- 将来自本地文件的 Nginx 配置文件添加到镜像中。
- 公开镜像的 80 端口。

这些 Nginx 配置文件是为了运行 sample 网站而配置的。将文件 nginx/global.conf 用 ADD 指令复制到“/etc/nginx/conf.d/”目录中。

④ 构建新的 Nginx 镜像。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！


```
# docker build -t jamtur01/nginx .
```

⑤ 展示 Nginx 镜像的构建历史。

```
[root@l-client sample]# docker history jamtur01/nginx
IMAGE          CREATED             CREATED BY                                      SIZE      COMMENT
25a2b32d2702   About a minute ago  /bin/sh -c #（nop）EXPOSE 80/tcp                0 B
705b78ae3e44   About a minute ago  /bin/sh -c #（nop）ADD file:d6698a182fafaf3cb0   415 B
0c0086998e24   About a minute ago  /bin/sh -c #（nop）ADD file:9778ae1b43896011cc   286 B
a0851da17b8f   2 minutes ago       /bin/sh -c sudo mkdir -p /var/www/html/websit   0 B
06efd6b163ac   2 minutes ago       /bin/sh -c sudo apt-get -yqq install nginx      18.13 MB
70e78a7f4121   35 minutes ago      /bin/sh -c #（nop）ENV REFRESHED_AT=2016-12-25  0 B
233522435d2c   40 minutes ago      /bin/sh -c #（nop）MAINTAINER James Turnbull "   0 B
ebdc8e295a2e   11 months ago       /bin/sh -c #（nop）CMD ["/bin/bash"]            0 B
<missing>      11 months ago       /bin/sh -c sed -i 's/^#\s*\（deb.*universe\）$/  1.895 kB
<missing>      11 months ago       /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
<missing>      11 months ago       /bin/sh -c #（nop）ADD file:7ce20ce3daa6af21db   187.7 MB
```

⑥ 下载 sample 网站测试网页。

```
[root@l-client sample]# mkdir website && cd website
[root@l-client website]# wget https://raw.githubusercontent.com/jamtur01/dockerbook-code/
master/code/5/sample/website/index.html
```

测试网页内容如下：

```
<head>
<title>Test website</title>
</head>
<body>
<h1>This is a test website</h1>
</body>
```

⑦ 运行第一个 Nginx Web 站点容器，并做数据卷映射。

```
[root@l-client sample]# docker run -d -p 80 --name website -v $PWD/website:/var/www/html/
website jamtur01/nginx nginx
c0bded9df5549010ad12ea8eb3d1dc0e92f12087e227ba3e82b15dee2c173b13
```

⑧ 查看网站容器。

```
[root@l-client sample]# docker ps -l
CONTAINER ID  IMAGE           COMMAND   CREATED         STATUS         PORTS                  NAMES
c0bded9df554  jamtur01/nginx  "nginx"   8 minutes ago   Up 8 minutes   0.0.0.0:32768->80/tcp  website
```

⑨ 浏览 sample 网站。

测试容器运行是否正确，运行结果如图 7-1-1 所示。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

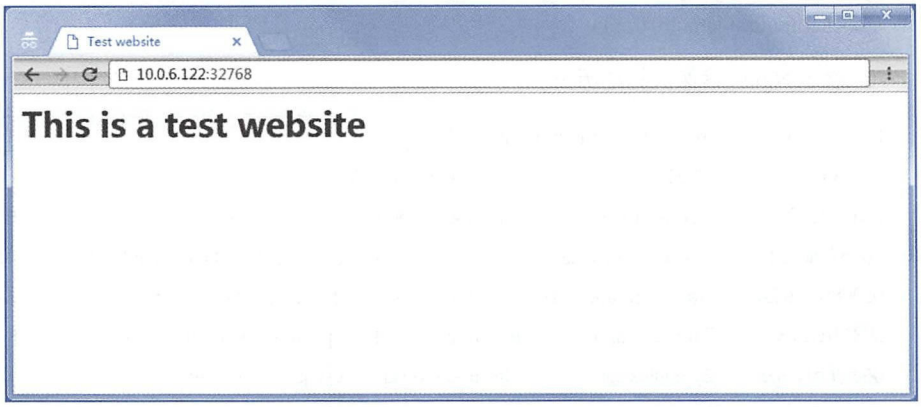


图 7-1-1
sample 网站
运行结果

⑩ 修改网站实例网页。

查看网页内容：

```
# vi $PWD/website/index.html
<head>
<title>Test website</title>
</head>
<body>
<h1>This is a test website</h1>
</body>
```

修改为：

```
<head>
<title>Test website</title>
</head>
<body>
<h1>This is a test website for docker</h1>
</body>
```

⑪ 重新刷新网站，查看网页内容变化，如图 7-1-2 所示。

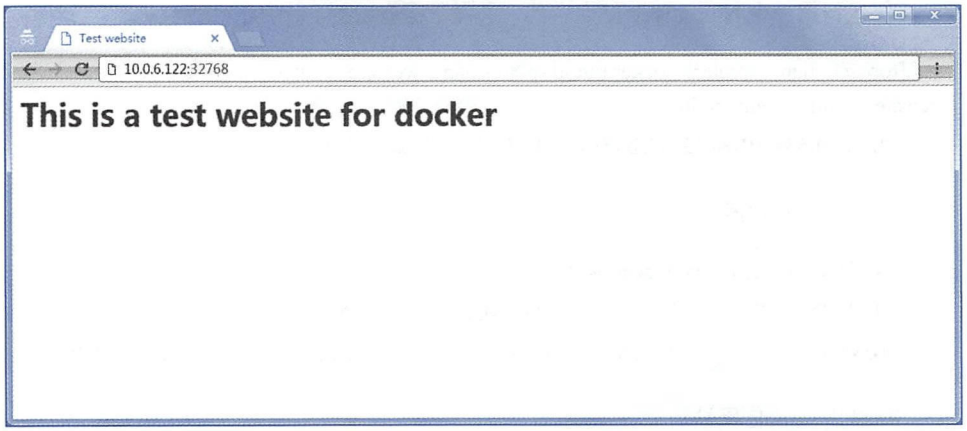


图 7-1-2
修改后的 sample
网站运行结果

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

项目实训

笔 记

【实训题目】

使用 Dockerfile 构建 Wordpress 应用。

【实训目的】

1. 了解 Dockerfile 的使用方法。
2. 理解 Dockerfile 的构建过程。
3. 掌握 WordPress 的部署思路。

【实训内容】

1. 部署 Docker 环境，关闭防火墙规则。
2. 下载 WordPress 的镜像包，或是将 Docker 环境联网。
3. 建立 WordPress 的工程目录，编写 Dockerfile 文件。
4. 使用 Dockerfile 构建 WordPress 应用。
5. 使用构建生成的镜像启动容器，通过网页访问验证。

任务 7.2　持续集成管理软件 Jenkins 应用

任务描述

1. 安装 Docker 环境，关闭防火墙规则。
2. 编写构建 Jenkins 服务的 Dockerfile 文件。
3. 构建 Jenkins 应用服务，并验证结果。

知识学习

1. Jenkins 介绍

Jenkins 是一个开源软件项目，旨在提供一个开放易用的软件平台，使软件的持续集成变成可能。Jenkins 是基于 Java 开发的一种持续集成工具，用于监控持续重复的工作，功能如下。

① 持续的软件版本发布/测试项目。

② 监控外部调用执行的工作。

2. Jenkins 的启动方式

首先保证系统中已经安装了 JDK，最好是 jdk1.5 以上。

第一种启动方法，切换到 jenkins.war 存放的目录，输入如下命令：

```
$ java -jar jenkins.war
```

然后在浏览器中（推荐用火狐）输入“localhost:8080”，localhost 可以是本机的 IP，也可以是计算机名，这样就可以打开 Jenkins。

第二种方法是用 Tomcat 打开：

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁传播，违者自负法律责任！

解压 Tomcat 到某个目录，如“/usr/local”，进入 Tomcat 下的“/bin”目录，启动 Tomcat，将 jenkins.war 文件放入 Tomcat 下的 webapps 目录下，启动 Jenkins 时，会自动在 webapps 目录下建立 jenkins 目录，所以在地址栏上需要输入的地址与上一种方法有点不一样。

任务实施

① 创建一个目录 jenkins，保存相关的配置信息和内容。

```
# cd
# mkdir jenkins
# cd jenkins/
```

② 定义 jenkins 的 Dockerfile。

```
# vi Dockerfile
FROM ubuntu:14.04.3
MAINTAINER james@example.com
ENV REFRESHED_AT 2016-12-25
RUN apt-get update -qq && apt-get install -qqy curl apt-transport-https
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:0 --recv-keys
58118E89F3A912897C070ADBF76221572C52609D
RUN echo deb https://apt.dockerproject.org/repo ubuntu
trusty main > /etc/apt/sources.list.d/docker.list
RUN apt-get update -qq && apt-get install -qqy iptables ca-certificates openjdk-7-jdk git-core
docker.io
ENV JENKINS_HOME /opt/jenkins/data
ENV JENKINS_MIRROR http://mirrors.jenkins-ci.org
RUN mkdir -p $JENKINS_HOME/plugins
RUN curl -f -o /opt/jenkins/jenkins.war -L $JENKINS_MIRROR/war-stable/latest/jenkins.war
RUN for plugin in chucknorris greenballs scm-api git-client git ws-cleanup; do curl -sf -o
$JENKINS_HOME/plugins/${plugin}.hpi-L $JENKINS_MIRROR/plugins/${plugin}/latest/
${plugin}.hpi; done
ADD ./dockerjenkins.sh /usr/local/bin/dockerjenkins.sh
RUN chmod +x /usr/local/bin/dockerjenkins.sh
VOLUME /var/lib/docker
EXPOSE 8080
ENTRYPOINT [ "/usr/local/bin/dockerjenkins.sh" ]
```

Dockerfile 的说明如下：

- jenkins 的 Dockerfile 继承了 ubuntu:14.0.4.3 镜像。
- apt-key 命令用于管理 Debian Linux 系统中的软件包密钥。每个发布的 deb 包都是通过密钥认证的，apt-key 用来管理密钥。
- 安装源加入需要的 docker apt 仓库，并安装执行 Docker 和 Jenkins 所需要的包。
- 创建“/opt/jenkins”目录，下载 jenkins.war 到这个目录，并下载 Jenkins 插件。
- 使用 ENV 指令把 JENKINS_HOME 和 JENKINS_MIRROR 环境变量设置为 Jenkins 的数据目录和镜像站点。
- 使用 VOLUME 指令指定/var/lib/docker 数据卷，运行容器的内部使用宿主机的文件系统作为容器的存储。这样容器的“/var/lib/docker”目录将保存在宿主机系统的

版权所有，严禁传播，违者自负法律责任！

“/var/lib/docker/volumes”目录下。

- Jenkins 默认对外的是 8080 端口。
- 最后指定要运行的 shell 脚本作为容器的启动命令。这个 shell 脚本帮助在宿主机上配置 Docker，允许在 Docker 容器里运行 Docker，开启 Docker 守护进程，并且启动 Jenkins。

③ 获取 dockerjenkins.sh 脚本。

```
#wget https://raw.githubusercontent.com/jamtur01/dockerbook-code/master/code/5/jenkins/dockerjenkins.sh
# chmod 0755 dockerjenkins.sh
```

④ 构建 docker-jnekins 镜像。

```
#docker build -t jamtur01/dockerjenkins
349da36a6408a03887eea50b37ffbc6ff06990287l5994221f71e8287df96015.
```

⑤ 运行 docker-jenkins 镜像。

```
#docker run -p 8080:8080 --name jenkins --privileged -djamtur01/dockerjenkins
```

⑥ 检查 docker jenkins 容器的日志。

```
# docker logs jenkins
/proc/self/fd /
/
docker: 'daemon' is not a docker command. See 'docker --help'.
Running from: /opt/jenkins/jenkins.war
webroot: EnvVars.masterEnvVars.get（"JENKINS_HOME"）
Dec 28, 2016 6:07:07 AM Main deleteWinstoneTempContents
WARNING: Failed to delete the temporary Winstone file /tmp/winstone/jenkins.war
Dec 28, 2016 6:07:10 AM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Logging initialized @4602ms
Dec 28, 2016 6:07:10 AM winstone.Logger logInternal
INFO: Beginning extraction from war file
```

⑦ 检查 Jenkins 的启动和执行。

```
Dec 28, 2016 6:09:10 AM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
```

项目实训

【实训题目】

基于 Rancher 管理平台部署 Jenkins 应用服务。

【实训目的】

1. 了解 Rancher 管理平台部署 Jenkins 应用服务的过程。
2. 比较管理平台部署的 Jenkins 和 dockerfile 构建的 Jenkins 有什么异同。

【实训内容】

1. 安装 Docker 环境，关闭防火墙策略。

版权所有，严禁传播，违者自负法律责任！

笔 记

2. 部署 Rancher 管理平台，单击节点即可。
3. 在 Rancher 管理平台上部署 Jenkins 应用服务。
4. 比较管理平台部署的 Jenkins 和 Dockerfile 构建的 Jenkins 的异同，形成说明文档。

任务 7.3　构建 Java Web 应用

任务描述

1. 构建一个镜像，负责下载 WAR 文件，并把 WAR 包保存在数据卷上。
2. 构建第二个镜像，负责运行 Tomcat，启动保持在共享数据卷上的 WAR 包。
3. 用刚才构建的镜像启动容器，查看效果。

知识学习

1. Tomcat 简介

Tomcat 是 Apache 软件基金会（Apache Software Foundation）的 Jakarta 项目中的一个核心项目，由 Apache、Sun 和其他一些公司及个人共同开发而成。由于有了 Sun 的参与和支持，最新的 Servlet 和 JSP 规范总是能在 Tomcat 中得到体现，Tomcat 5 支持最新的 Servlet 2.4 和 JSP 2.0 规范。因为 Tomcat 技术先进、性能稳定，而且免费，因而深受 Java 爱好者的喜爱并得到了部分软件开发商的认可，成为目前比较流行的 Web 服务器。

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试 JSP 程序的首选。对于一个初学者来说，可以这样认为，当在一台机器上配置好 Apache 服务器时，可利用它响应 HTML（标准通用标记语言下的一个应用）页面的访问请求。实际上，Tomcat 是 Apache 服务器的扩展，但运行时它是独立运行的，所以当开发者运行 Tomcat 时，它实际上是作为一个与 Apache 独立的进程单独运行的。

诀窍是，当配置正确时，Apache 为 HTML 页面服务，而 Tomcat 实际上运行 JSP 页面和 Servlet。另外，Tomcat 和 IIS 等 Web 服务器一样，具有处理 HTML 页面的功能。另外，它还是一个 Servlet 和 JSP 容器，独立的 Servlet 容器是 Tomcat 的默认模式。不过，Tomcat 处理静态 HTML 的能力不如 Apache 服务器。目前 Tomcat 最新版本为 9.0。

2. 目前常用的 Web 服务器

除了 Tomcat 和 Nginx 是比较热门的 Web 服务器之外，目前还有很多常用的 Web 服务器，如下所示。

① Apache 是世界使用排名第一的 Web 服务器软件。它可以运行在几乎所有广泛使用的计算机平台上。Apache 源于 NCSAhttpd 服务器，经过多次修改，成为世界上最流行的 Web 服务器软件之一。Apache 取自“a patchy server”的读音，意思是充满补丁的服务器，因为它是自由软件，所以不断有人来为它开发新的功能、新的特性、修改原来的缺陷。Apache 的特点是简单、速度快、性能稳定，并可作为代理服务器来使用。

② IIS 是英文 Internet Information Server 的缩写，译成中文就是“Internet 信息服务”。它是微软公司主推的服务器，最新的版本是 Windows2008 里面包含的 IIS 7。IIS 与

版权所有，严禁传播，违者自负法律责任！

Window Server 完全集成在一起，因而用户能够利用 Windows Server 和 NTFS（NT File System，NT 文件系统）内置的安全特性建立强大、灵活而安全的 Internet 和 Intranet 站点。

笔记

③ GFEGoogle 的 Web 服务器，用户数量激增，目前紧逼 IIS。

④ Lighttpd 是由德国人 Jan Kneschke 领导开发的，基于 BSD 许可的开源 Web 服务器软件，其根本的目的是提供一个专门针对高性能网站的，安全、快速、兼容性好且灵活的 Web Server 环境。它具有非常低的内存开销，CPU 占用率低，效能好，以及丰富的模块等特点。Lighttpd 是众多 OpenSource 轻量级的 Web Server 中较为优秀的一个，支持 FastCGI、CGI、Auth、输出压缩（Output Compress），URL 重写、Alias 等重要功能。

⑤ Zeus 是一个运行于 UNIX 下的非常优秀的 Web Server，据说性能超过 Apache，是效率最高的 Web Server 之一。

⑥ Sun 的 Java 系统 Web 服务器也就是以前的 Sun ONE Web Server，主要出现在那些运行 Sun 的 Solaris 操作系统的关键任务级 Web 服务器上。它最新的版本号是 6.1，可以支持 x86 版本 Solaris、Red Hat Linux、HP-UX 11i、 IBM AIX，甚至可以支持 Windows，但它的大多数用户都选择了 SPARC 版本的 Solaris 操作系统。

⑦ Resin 提供了最快的 JSP/Servlet 运行平台。在 Java 和 Javascript 的支持下，Resin 可以为任务灵活选用合适的开发语言。Resin 的一种先进的语言 XSL（XML Stylesheet Language）可以使得形式和内容相分离。

⑧ Jetty 是一个开源的 servlet 容器，它为基于 Java 的 Web 内容提供运行环境，例如 JSP 和 Servlet。Jetty 是使用 Java 语言编写的，它的 API 以一组 JAR 包的形式发布。开发人员可以将 Jetty 容器实例化成一个对象，可以迅速为一些独立运行（Stand-Alone）的 Java 应用提供网络和 Web 连接。

⑨ BEA WebLogic 是用于开发、集成、部署和管理大型分布式 Web 应用、网络应用和数据库应用的 Java 应用服务器，将 Java 的动态功能和 Java Enterprise 标准的安全性引入大型网络应用的开发、集成、部署和管理之中。 BEA WebLogic Server 拥有处理关键 Web 应用系统问题所需的性能、可扩展性和高可用性。

任务实施

（1）负责下载 WAR 文件的镜像创建步骤

① 创建保存目录和 Dockerfile。

```
# mkdir fetcher
# cd fetcher/
# touch Dockerfile
```

② 通过 wget 获取 WAR 文件，具体下载地址由用户决定。

```
# vi Dockerfile
FROM ubuntu:14.04.3
MAINTAINER James Turnbull "james@example.com"
ENV REFRESHED_AT 2016-12-25
RUN apt-get -y update
RUN apt-get -y install wget
VOLUME [ "/var/lib/tomcat7/webapps/" ]
```

版权所有，严禁传播，违者自负法律责任！


```
WORKDIR /var/lib/tomcat7/webapps/
ENTRYPOINT [ "wget" ]
CMD [ "-?" ]
```

当执行容器时，指定 URL 下载 WAR 包，并保存在/var/lib/tomcat7/webapps/目录下。此目录也是一个卷和所有容器的工作目录，然后把这个卷共享给运行 Tomcat 服务的容器。

③ 构建镜像。

```
# docker build -t jamtur01/fetcher .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu:14.04.3
 ---> ebdc8e295a2e
Step 2 : MAINTAINER James Turnbull "james@example.com"
 ---> Using cache
 ---> 233522435d2c
Step 3 : ENV REFRESHED_AT 2016-12-25
 ---> Using cache
 ---> 70e78a7f4121
Step 4 : RUN apt-get -y update
 ---> Running in bda317f06643
---> f35531a71f08
Removing intermediate container bda317f06643
Step 5 : RUN apt-get -y install wget
 ---> Running in 113c57bc332c
---> 0771b0536da9
Removing intermediate container 113c57bc332c
Step 6 : VOLUME /var/lib/tomcat7/webapps/
 ---> Running in d3d9221f691c
 ---> 4a5614f493ee
Removing intermediate container d3d9221f691c
Step 7 : WORKDIR /var/lib/tomcat7/webapps/
 ---> Running in 783910b7770c
 ---> db08b2409136
Removing intermediate container 783910b7770c
Step 8 : ENTRYPOINT wget
 ---> Running in 1cd35e6d9301
 ---> 3b3307763663
Removing intermediate container 1cd35e6d9301
Step 9 : CMD -?
 ---> Running in ae8f6fb31831
 ---> c74a890eddd5
Removing intermediate container ae8f6fb31831
Successfully built c74a890eddd5
```

④ 运行容器，并指定 URL 下载 WAR 包。

```
# docker run -t -i --name sample jamtur01/fetcher https://tomcat.apache.org/tomcat-7.0-doc/
appdev/ sample/sample.war https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war
--2016-12-29 07:47:29--  https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war
```

```
Resolving tomcat.apache.org （tomcat.apache.org）... 88.198.26.2, 140.211.11.105,
2a01:4f8:130:2192::2
Connecting to tomcat.apache.org （tomcat.apache.org）|88.198.26.2|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4606 （4.5K）
Saving to: 'sample.war'

100%[=======================>] 4,606          --.-K/s    in 0s

2016-12-29 07:47:34 （103 MB/s）- 'sample.war' saved
[4606/4606]

--2016-12-29 07:47:34--  https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war
Reusing existing connection to tomcat.apache.org:443.
HTTP request sent, awaiting response... 200 OK
Length: 4606 （4.5K）
Saving to: 'sample.war.1'

100%[=======================>] 4,606          3.64KB/s    in 1.2s

2016-12-29 07:47:36 （3.64 KB/s）- 'sample.war.1' saved [4606/4606]

FINISHED --2016-12-29 07:47:36--
Total wall clock time: 6.4s
Downloaded: 2 files, 9.0K in 1.2s （7.29 KB/s）
```

容器运行，指定下载 URL 地址为 https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war；下载目录为容器的工作目录：var/lib/tomcat7/webapps/。

⑤ 通过 docker inspect 命令查找卷的存储位置。

```
# docker inspect -f "{{ range .Mounts }} {{.}} {{end}}" sample
  {dc4127683406b0334ec83c37c356b357fbefdf1608cb916491a6a7b091a71b53 /var/lib/docker/
volumes/dc4127683406b0334ec83c37c356b357fbefdf1608cb916491a6a7b091a71b53/_data /var/
lib/ tomcat7/webapps local  true }
```

⑥ 可以通过 ls 查看卷所在的目录。

```
# ls -l /var/lib/docker/volumes/
```

（2）负责运行 Tomat 和 Web 应用的镜像创建步骤

① 创建 Tomcat7 Dockerfile。

```
#mkdir tomcat7
#cd tomcat7/
#vi Dockerfile
FROM ubuntu:14.04.3
MAINTAINER James Turnbull james@example.com
ENV REFRESHED_AT 2016-12-25
RUN apt-get -y update
RUN apt-get -y install tomcat7 default-jdk
```

```
ENV CATALINA_HOME /usr/share/tomcat7
ENV CATALINA_BASE /var/lib/tomcat7
ENV CATALINA_PID /var/run/tomcat7.pid
ENV CATALINA_SH /usr/share/tomcat7/bin/catalina.sh
ENV CATALINA_TMPDIR /tmp/tomcat7-tomcat7-tmp
RUN mkdir -p $CATALINA_TMPDIR
VOLUME [ "/var/lib/tomcat7/webapps/" ]
EXPOSE 8080
ENTRYPOINT [ "/usr/share/tomcat7/bin/catalina.sh","run" ]
```

这个镜像很简单，首先安装 Java JDK 和 Tomcat 服务器，接着设置环境变量，然后创建了/var/lib/tomcat7/webapps/卷，公开 Tomcat 默认的 8080 端口，最后 ENTRYPOINT 指定启动 Tomcat 的命令。

② 构建 Tomcat7 镜像。

```
# docker build -t jamtur01/tomcat7 .
….
…. done.
Processing triggers for sgml-base（1.26+nmu4ubuntu1）...
 ---> 6bb373b73ddc
Removing intermediate container 6a87159cda44
Step 6 : ENV CATALINA_HOME /usr/share/tomcat7
 ---> Running in dda79e0e5eca
 ---> 6f847ea986b3
Removing intermediate container dda79e0e5eca
Step 7 : ENV CATALINA_BASE /var/lib/tomcat7
 ---> Running in e05301f878dc
 ---> 297d325d22ee
Removing intermediate container e05301f878dc
Step 8 : ENV CATALINA_PID /var/run/tomcat7.pid
 ---> Running in 64bf2b71eb25
 ---> 1cf2c783758d
Removing intermediate container 64bf2b71eb25
Step 9 : ENV CATALINA_SH /usr/share/tomcat7/bin/catalina.sh
 ---> Running in 3bf3f9cff57f
 ---> 3b766d1471e1
Removing intermediate container 3bf3f9cff57f
Step 10 : ENV CATALINA_TMPDIR /tmp/tomcat7-tomcat7-tmp
 ---> Running in c25a6b890171
 ---> 96b9faaae6f5
Removing intermediate container c25a6b890171
Step 11 : RUN mkdir -p $CATALINA_TMPDIR
 ---> Running in b9d820b71657
 ---> 0c5ff83b4fab
Removing intermediate container b9d820b71657
Step 12 : VOLUME /var/lib/tomcat7/webapps/
 ---> Running in b0b7cdc08643
 ---> 2d5779167f81
Removing intermediate container b0b7cdc08643
Step 13 : EXPOSE 8080
```

```
 ---> Running in 9a9fdeaa5f02
 ---> df0202997162
Removing intermediate container 9a9fdeaa5f02
Step 14 : ENTRYPOINT /usr/share/tomcat7/bin/catalina.sh run
 ---> Running in cd1dd633cc90
 ---> 6b15a1f76a5f
Removing intermediate container cd1dd633cc90
Successfully built 6b15a1f76a5f
```

③ 创建第一个 Tomcat 实例。

```
# docker run --name sample_app --volumes-from sample -d -P jamtur01/tomcat7
c12eca5093ab403635ac44c9335a5954cedf63732f04ff016a6239bd0f04df39
```

④ 查找 Tomcat 应用的端口。

```
# docker port sample_app 8080
0.0.0.0:32798
```

项目实训

【实训题目】

在 Tomcat 服务器中启动 Jenkins 应用服务。

【实训目的】

1. 掌握 Tomcat 的部署、使用。
2. 掌握 Jenkins 的部署、使用。
3. 比较和容器启动 Jenkins 服务的异同。

【实训内容】

1. 安装 CentOS 7 系统环境。
2. 安装 JDK 环境，部署 Tomcat 服务器。
3. 部署 Jenkins 服务并启动。
4. 比较和容器启动 Jenkins 服务的异同。

任务 7.4　构建 Docker 负载均衡

任务描述

1. 构建 WordPress 应用服务。
2. 增加一个 WordPress 应用服务的负载均衡。
3. 进行 WordPress 应用扩展。
4. 访问 WordPress 应用服务，验证负载均衡的效果。

知识学习

1. 弹性伸缩的简介

弹性伸缩（Auto Scaling）根据业务需求和伸缩策略，为用户自动调整计算资源。用

版权所有，严禁传播，违者自负法律责任！

笔 记

户可设置定时、周期或监控策略，恰到好处地增加或减少 CVM 实例，并完成实例配置，保证业务平稳健康运行。在需求高峰期时，弹性伸缩自动增加 CVM 实例的数量，以保证性能不受影响；当需求较低时，则会减少 CVM 实例数量以降低成本。弹性伸缩既适合需求稳定的应用程序，同时也适合每天、每周、每月使用量波动的应用程序。

可伸缩性（可扩展性）是一种对软件系统计算处理能力的设计指标。高可伸缩性代表一种弹性，在系统扩展成长过程中，软件能够保证旺盛的生命力，通过很少的改动甚至只是硬件设备的添置，就能实现整个系统处理能力的线性增长，实现高吞吐量和低延迟高性能。

可伸缩性和纯粹性能优化有本质区别，可伸缩性是高性能、低成本和可维护性等诸多因素的综合考量和平衡，讲究平滑线性的性能提升，更侧重于系统的水平伸缩，通过廉价的服务器实现分布式计算；而普通性能优化只是单台机器的性能指标优化。它们的共同点都是根据应用系统的特点在吞吐量和延迟之间进行一个侧重选择，当然水平伸缩分区后会带来 CAP 定理约束。

2. 负载均衡的实现方法

为了提高服务器的性能和工作负载能力，公司通常会使用 DNS 服务器、网络地址转换等来实现多服务器负载均衡，特别是目前企业对外的互联网 Web 网站，许多都是通过几台服务器来完成服务器访问的负载均衡。目前，企业使用的所谓负载均衡服务器，实际上是应用系统的一种控制服务器，所有用户的请求都首先到此服务器，然后由此服务器根据各个实际处理服务器的状态将请求具体分配到某个实际处理服务器中，对外公开的域名与 IP 地址都是这台服务器的。负载均衡控制与管理软件安装在这台服务器上，这台服务器一般只对负载均衡任务分配，但实际不对网络请求进行处理。

（1）企业实现 Web 服务器负载均衡

为了将负载均匀地分配给内部的多个服务器上，就需要应用一定的负载均衡策略。通过服务器负载均衡设备实现各服务器群的流量动态负载均衡，并互为冗余备份。新系统应有一定的扩展性，如果数据访问量继续增大，则可再添加新的服务器加入负载均衡系统。对于 Web 服务应用，同时有几台机器提供服务，每台机器的状态可以设为 regular（正常工作）或 backup（备份状态），或者同时设定为 regular 状态。负载均衡设备根据管理员事先设定的负载算法和当前网络实际的动态负载情况决定下一个用户的请求将被重定向到的服务器。而这一切对于用户来说是完全透明的，用户完成了对 Web 服务的请求，并不用关心具体是哪台服务器完成的。

（2）使用网络地址转换实现多服务器负载均衡

在支持负载均衡的地址转换网关中可以将一个外部 IP 地址映射为多个内部 IP 地址，对每次的 TCP 连接请求动态使用其中一个内部地址，达到负载均衡的目的。很多硬件厂商将这种技术集成在他们的交换机中，作为他们第四层交换的一种功能来实现，一般通过随机选择、根据服务器的连接数量或者响应时间进行选择的负载均衡策略来分配负载。然而硬件实现的负载控制器灵活性不强，不能支持更优化的负载均衡策略和更复杂的应用协议。

基于网络地址转换的负载均衡器可以有效地解决服务器端的 CPU 和磁盘 I/O 负载，然而负载均衡器本身的性能受网络 I/O 的限制，在一定硬件条件下具有一定的带宽限制，但可以通过改善算法和提高运行负载均衡程序的硬件性能来提高这个带宽限制。不同的服务类型可对不同的服务器资源进行占用，开发者使用的负载衡量策略是使用同一个负载进行评估，

版权所有，严禁传播，违者自负法律责任！

这对于大多数条件是适合的，然而最好的办法是针对不同的资源，如 CPU、磁盘 I/O 或网络 I/O 等，分别监视服务器负载，由中心控制器选择最合适的服务器分发客户请求。

笔 记

（3）使用 DNS 服务器实现负载均衡

如果访问企业网服务器的用户急剧增加，一台服务器难以满足用户的访问需要，那么如何才能保证用户的正常访问呢？解决方法有很多，如使用 Windows 2000 或 Windows Server 2003 提供网络负载均衡服务，但该服务的设置非常复杂。而通过 DNS 服务器实现网络负载均衡则是一种比较简单的方法。企业网通常由很多子网构成，为了降低网络中的数据流量，客户机最好能访问处于同一子网内的 Web 服务器。虽然实现了网络负载均衡功能，但并不能保证客户访问的是本子网的 Web 服务器。其实这个问题也很好解决，只要启用 DNS 服务器的网络掩码排序功能即可。在 DNS 管理器窗口中，右键单击 DNS 服务器，在打开的快捷菜单中选择“属性”命令，然后在“属性”对话框中切换到“高级”选项卡，勾选“服务器”选项列表框中的“启用网络掩码排序”复选框即可。这样客户机每次都能访问到本子网内的 Web 服务器了。完成以上设置后，就实现了网络负载均衡功能，把客户的访问分担到每个 Web 服务器上，并且还减少了跨子网的网络通信流量，大大降低了企业网的通信负担。

（4）企业实现 SQL Server 数据库服务器负载均衡

数据库服务器可以说是应用范围最广的数据库产品，并且越来越多地在大型和比较关键的应用系统中提供服务。当企业应用越来越复杂、数据量越来越大的时候，SQL Server 数据库要不停地进行处理、存储、查询的工作，这个时候企业就要考虑 SQL Server 数据库服务器的性能和速度及安全性了。然而，长期以来，SQL Server 数据库服务器都只有热备的解决方案，而没有负载均衡和集群的解决方案。

随着数据库路由器软件 ICX 的出现，为基于 MS SQL Server 的数据库系统提供了一种更优秀的集群解决方案。它可以真正地实现 SQL Server 数据库服务器的动态负载均衡，提高性能和速度；它可以真正地为 SQL Server 数据库服务器不间断地提供服务，在服务器发生故障的时候实时切换到其他服务器上继续提供服务，切换时间为零。数据库路由器是实时并发数据库事务处理的同步复制器和负载均衡器。所有的数据库客户都通过 ICX 访问数据库。当访问、查询 SQL Server 数据库的时候，ICX 可以根据实际情况分配服务器来提供服务，大大提高服务速度和优化性能，完成服务器负载均衡。ICX 可以同时连接多台数据库，这若干台数据库的内容在任何时刻都由 ICX 保证是完全一致的。也就是说，ICX 采用了全新的并发事务处理的方式向连接的 N 台数据库同步复制事务处理，使得系统在任何时刻都具有多个一致的最新逻辑数据库数据集。当其中一台数据库服务器发生故障的时候，ICX 可以实时地切换到其他服务器上来继续提供服务，真正地实现零时间的服务器切换，大大提高安全性，真正意义地实现服务器不间断服务。

任务实施

下面介绍基于 Rancher 实现的负载均衡与弹性扩展架构。

（1）启动一个 WordPress 服务

① 登录 Rancher 管理平台，单击“应用商店”按钮进入页面，找到 WordPress 应用，如图 7-4-1 所示。

版权所有，严禁传播，违者自负法律责任！

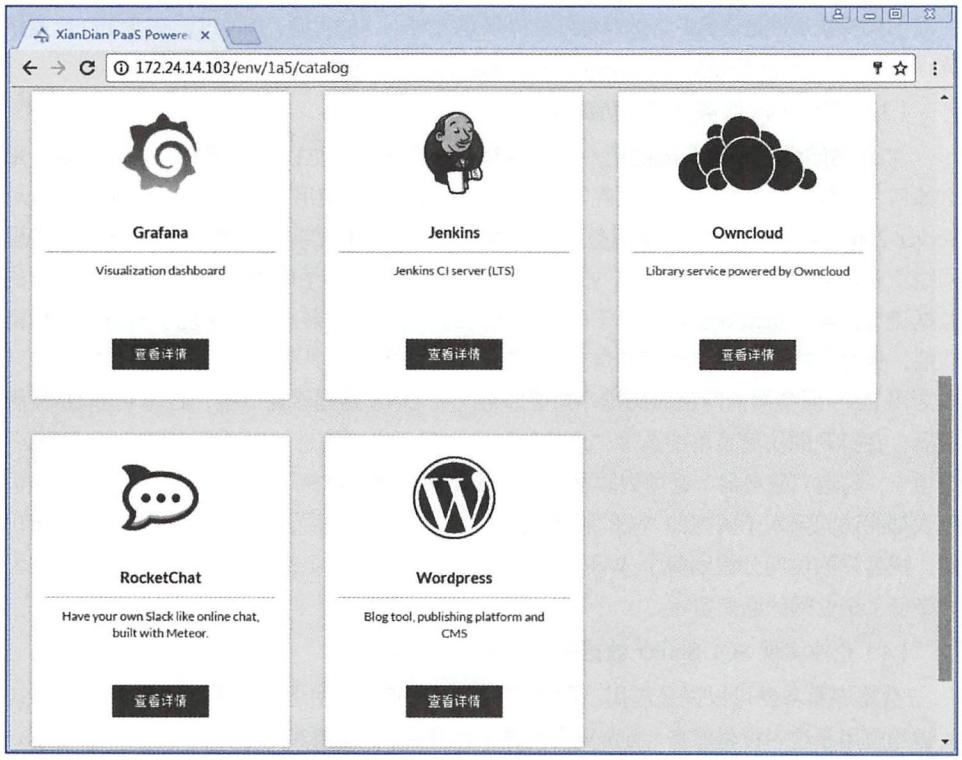


图 7-4-1
WordPress 应用

② 单击“查看详情”按钮，进行应用模板部署，在“名称”文本框中输入“wordpress”，在“Public Port”文本框中输入“80”，单击“启动”按钮开始创建，如图 7-4-2 所示。

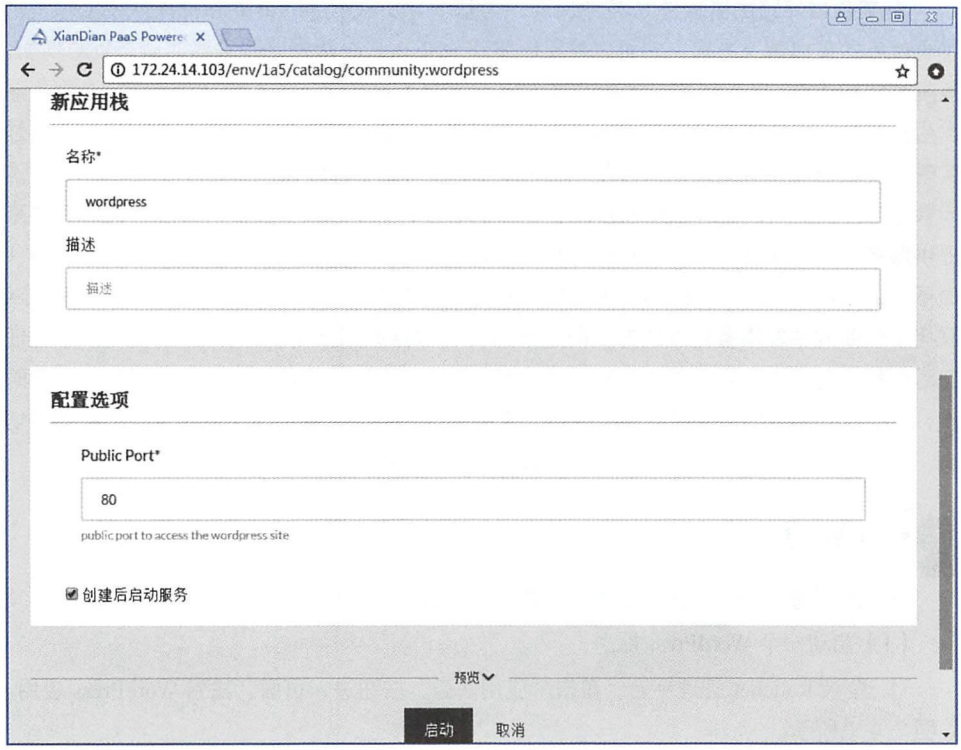


图 7-4-2
部署 WordPress

版权所有，严禁传播，违者自负法律责任！

③ 创建成功后，返回的结果如图 7-4-3 所示。

图 7-4-3
创建成功后
返回的结果

（2）添加一台扩展容器

① 在管理平台上单击“应用栈”，找到 WordPress 应用库，单击“wordpress”应用，WordPress 主页如图 7-4-4 所示。

图 7-4-4
WordPress 主页

② 单击数量旁边的“+”按钮，添加一台扩展容器，操作如图 7-4-5 所示。

版权所有，严禁传播，违者自负法律责任！

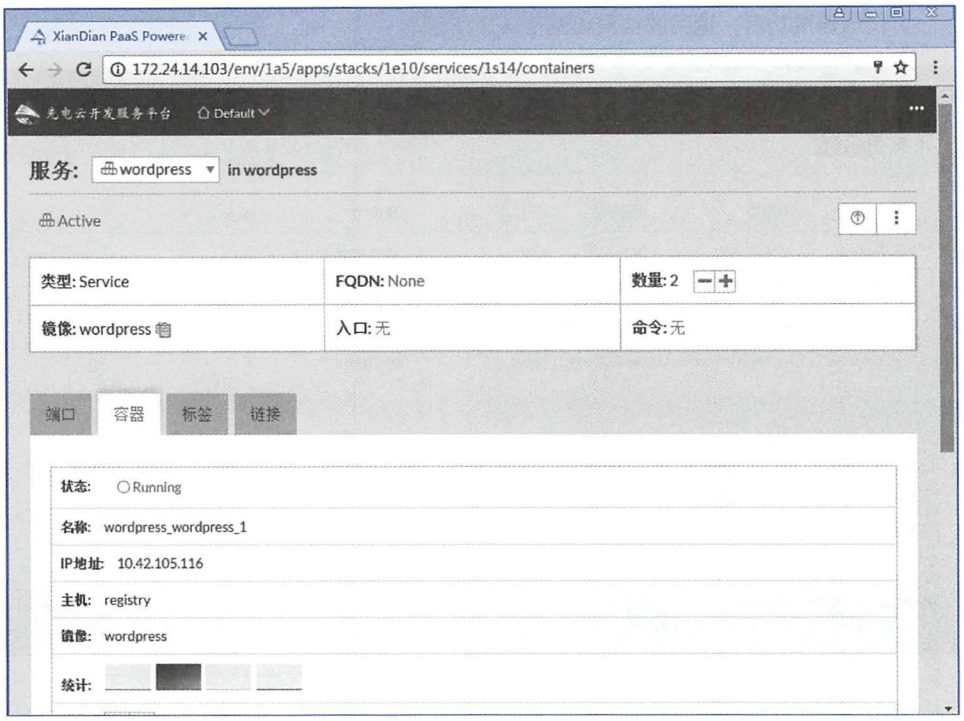


图 7-4-5
应用添加

③ 添加成功后的结果反馈如图 7-4-6 所示。

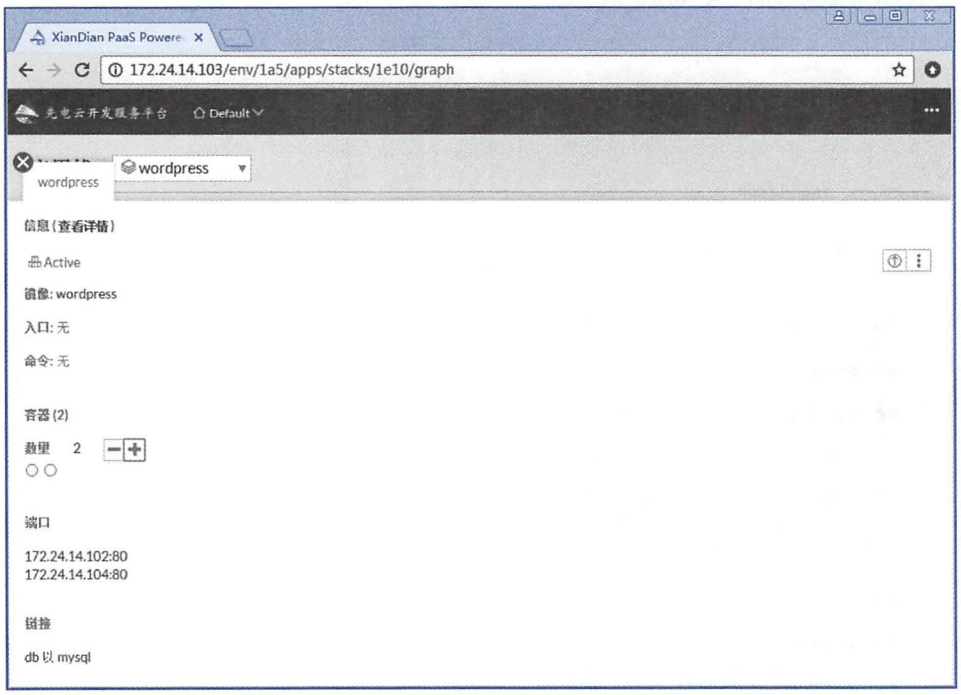


图 7-4-6
添加成功后的
结果反馈

(3) 在应用栈中设置负载均衡

① 进入管理平台，找到 WordPress 应用，单击“添加服务”按钮，在打开的下拉列表中选择“添加负载均衡”选项，如图 7-4-7 所示。

版权所有，严禁传播，违者自负法律责任！

图 7-4-7
选择“添加负载均衡”选项

② 进入添加负载均衡界面，输入名称、源端口、目标端口，可根据自己的实际情况输入，如图 7-4-8 所示。

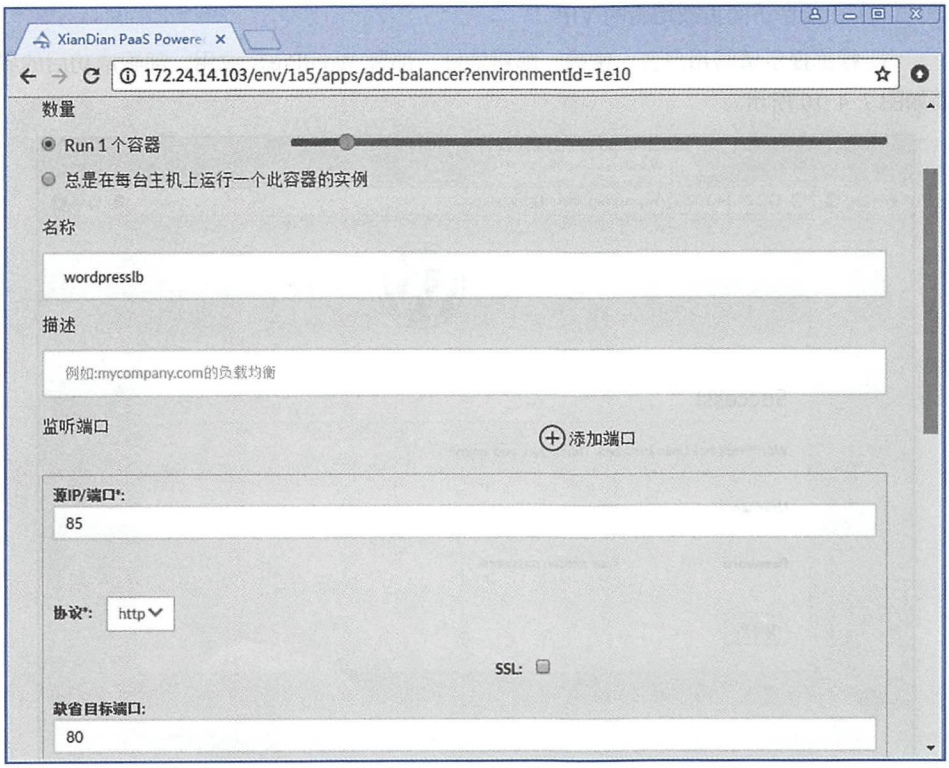


图 7-4-8
配置负载均衡

版权所有，严禁传播，违者自负法律责任！

③ 部署成功后的结果如图 7-4-9 所示。

图 7-4-9
成功启动负载均衡后的结果

（4）在前台访问负载均衡的 VIP

① 根据提示填写用户名、密码、邮箱地址、部署 WordPress 应用，部署成功后的界面如图 7-4-10 所示。

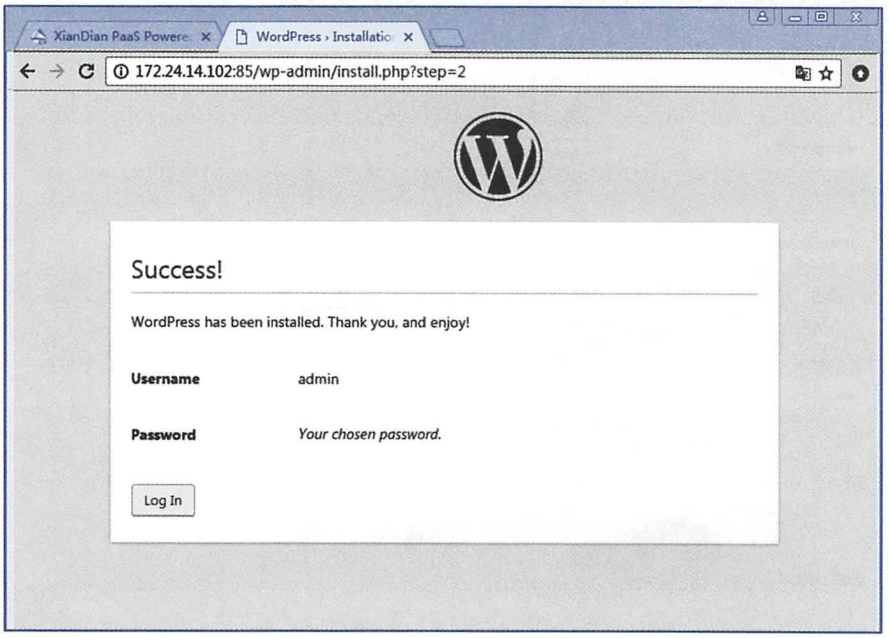


图 7-4-10
部署成功后的界面

版权所有，严禁传播，违者自负法律责任！

② 根据用户名和密码直接访问 WordPress 的站点，发布第一篇博客，如图 7-4-11 所示。

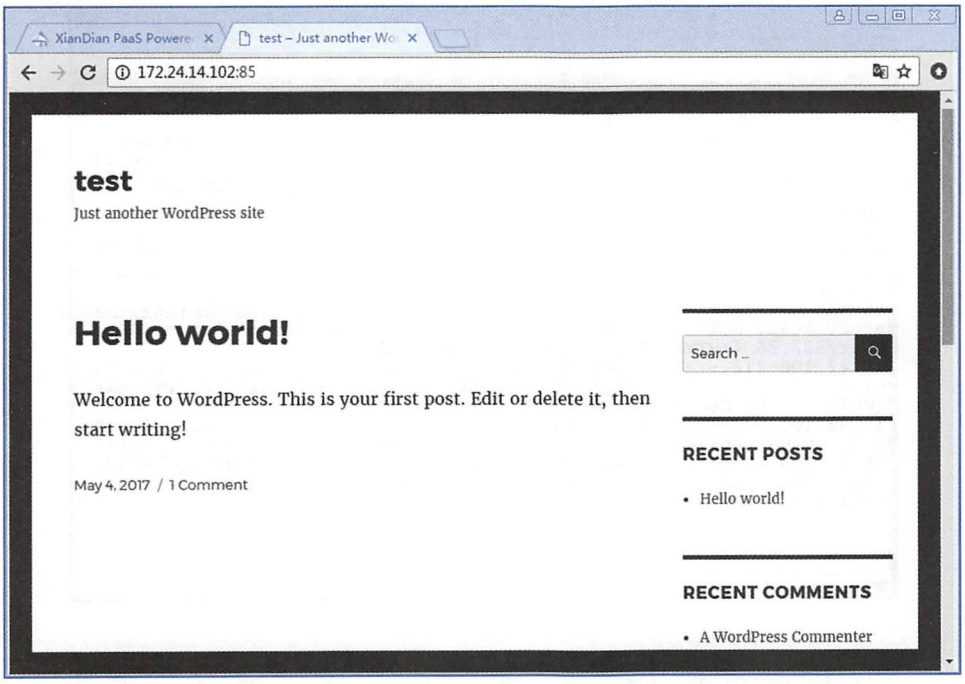


图 7-4-11
博客发布成功

(5) 观察两个容器中的日志状态

① 访问 WordPress 主页，查看发表的第一篇博客，并刷新，如图 7-4-12 所示。

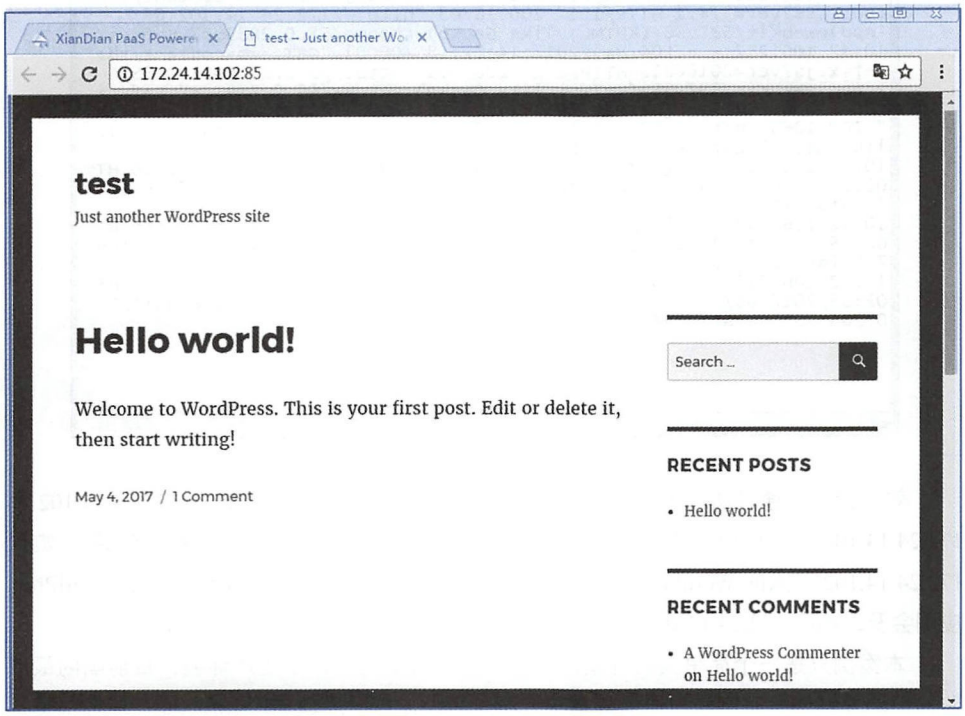


图 7-4-12
查看并刷新博客主页

版权所有，严禁传播，违者自负法律责任！

② 使用 CRT 远程连接工具连接到 Rancher 管理平台的两个 host 节点，并通过 docker logs 命令查看节点日志信息，操作结果如图 7-4-13 和图 7-4-14 所示。

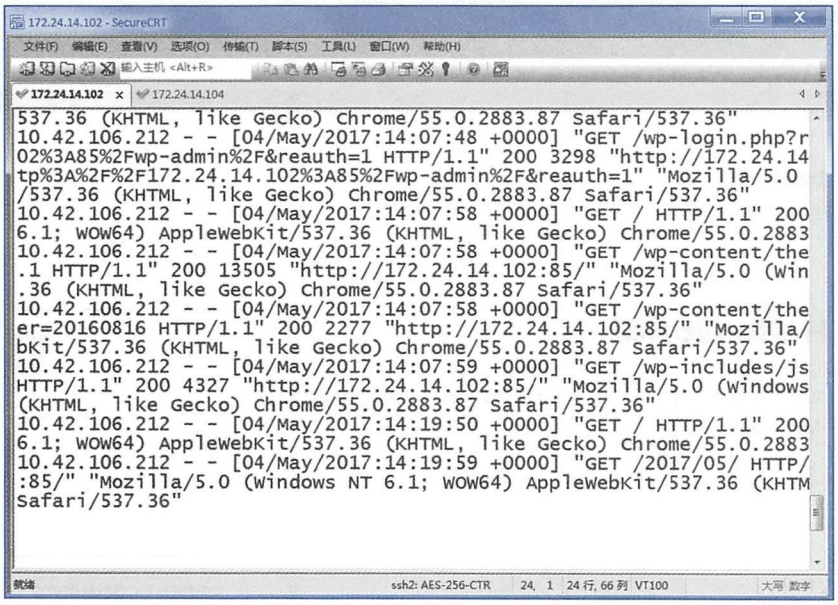


图 7-4-13
hosts 日志 1

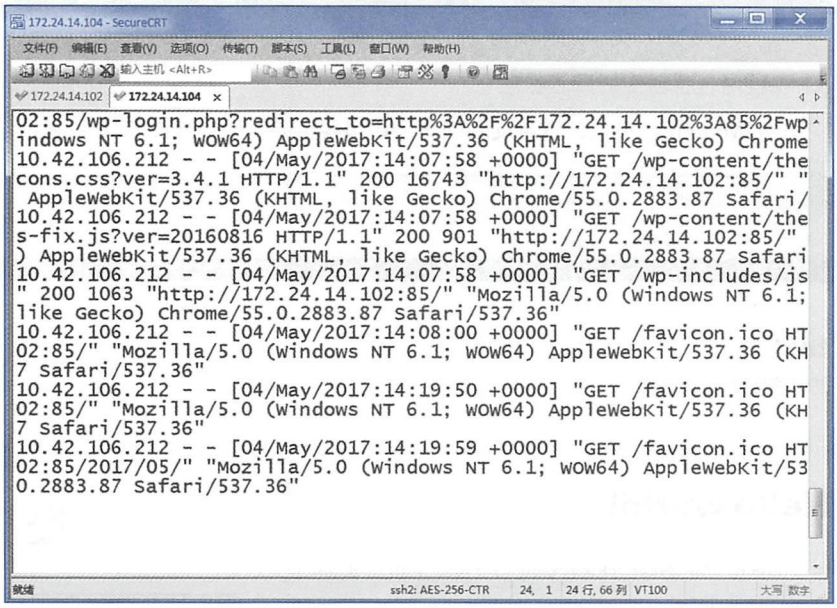


图 7-4-14
hosts 日志 2

效果分析：通过访问 172.24.14.102 主机上的 WordPress 的主页，在 172.24.14.102 和 172.24.14.104 节点同时收到 172.24.14.102 的请求，达到了负载均衡的效果，如果 172.24.14.102 节点的 WordPress 应用因为某种原因挂掉了，172.24.14.104 节点的 WordPress 应用会及时响应 172.24.14.102 主机的请求，保证 WordPress 服务的正常运行。

本案例只是一个简单实验性案例，并不能大批量地运用于生产环境，目前国内最成熟的自动弹性伸缩案例是阿里云的解决方案，包括监控时序数据库 Influxdb、监控报警框

版权所有，严禁传播，违者自负法律责任！

架 Kapacitor 及界面展现 Grafana。阿里云容器服务会通过监控报警框架 Kapacitor 自动采集容器的监控数据，并可以通过集成将监控数据发送到监控框架中。有了监控数据，开发者就可以在监控框架中定义自己的报警规则，当指标发生报警的时候调用阿里云容器服务提供的触发器来进行容器的扩容或者缩容。

所有的监控信息都会存放到监控时序数据库 Influxdb 中，并通过界面展现 Grafana 来进行很好的人机交流。

项目实训

【实训题目】

通过增加 Nginx 的权值实现应用服务的负载均衡。

【实训目的】

1. 熟悉 Nginx 应用服务的部署。
2. 了解负载均衡的实现原理。
3. 掌握通过增加 Nginx 的权值实现 Nginx 服务的负载均衡的方法。

【实训内容】

1. 安装 CentOS 7 的操作系统，准备 Docker 实训环境。
2. 配置 yum 源，添加 Nginx 的 yum 源文件。
3. 在 CentOS 7 系统上安装 Nginx 服务，启动并网页验证。
4. 启动两个 Nginx 容器，映射不同的端口。
5. 创建两个工作目录，存放不同的 Web 页面。
6. 配置 Nginx 的权值，分别指向两个容器的 URL，注意端口。
7. 查看 Nginx 服务，验证负载均衡效果。

单元小结

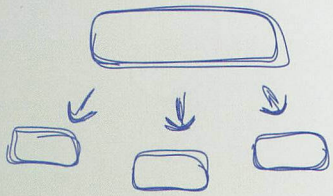
本单元通过构建 Nginx Web 应用、Jenkins 应用部署 Tomcat 服务器，学生了解了 Dockerfile 的编写方式和语法规则，深入理解了 Dockerfile 构建应用的过程。通过在 Docker 集群中进行负载均衡的设计，进一步加深学生对弹性扩展和负载均衡的理解。经过本单元任务的实现，锻炼了学生的项目开发能力和实践能力，对于以后接手这方面的工作有很大的帮助。

参考文献

[1] James Turnbull. THE DOCKER BOOK（2 版）. 北京：人民邮电出版社，2016.
[2] 华为 Docker 实践小组. Docker 进阶与实战. 北京：机械工业出版社，2016.
[3] 戈阿冈. Docker 经典案例. 北京：人民邮电出版社，2017.
[4] 孙宏亮. Docker 源码分析. 北京：机械工业出版社，2015.

CETC 云计算技术与应用专业校企合作系列教材

- Android云存储客户端开发
- 软件定义网络（SDN）技术与应用
- ✔ Docker容器技术与应用
- 云存储技术与应用
- Hadoop大数据平台构建与应用
- 云计算数据中心运维
- 虚拟化技术与应用
- 云计算网络技术与应用
- Python编程实战



ISBN 978-7-04-048943-9

定价 29.00 元